# Apache Solr Reference Guide

## Covering Apache Solr 4.10

# Apache Solr Reference Guide

This reference guide describes Apache Solr, the open source solution for search. You can download Apache Solr from the Solr website at http://lucene.apache.org/solr/.

This Guide contains the following sections:

**Getting Started**: This section guides you through the installation and setup of Solr.

**Using the Solr Administration User Interface**: This section introduces the Solr Web-based user interface. From your browser you can view configuration files, submit queries, view logfile settings and Java environment settings, and monitor and control distributed configurations.

**Documents, Fields, and Schema Design**: This section 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.

**Understanding Analyzers, Tokenizers, and Filters**: This section 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.

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

**Searching**: This section 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.

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

**Managing Solr**: This section discusses important topics for running and monitoring Solr. It describes running Solr in the Apache Tomcat servlet runner and Web server. Other topics include how to back up a Solr instance, and how to run Solr with Java Management Extensions (JMX).

**SolrCloud**: This section describes the newest and most exciting of Solr's new features, SolrCloud, which provides comprehensive distributed capabilities.

**Legacy Scaling and Distribution**: This section 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.

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

# About This Guide

This guide describes all of the important features and functions of Apache Solr. It is free to download from http://lucene.apache.org/solr/.

Designed to provide high-level documentation, this 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 life cycle, for whenever you need authoritative information about Solr.

The material as presented assumes that you are 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.

## Special Inline Notes

Special notes are included throughout these pages.

| Note Type | Look & Description |
|---|---|
| **Information** | ⓘ Notes with a blue background are used for information that is important for you to know. |
| **Notes** | ⚠ Yellow notes are further clarifications of important points to keep in mind while using Solr. |
| **Tip** | ✓ Notes with a green background are Helpful Tips. |
| **Warning** | ❗ Notes with a red background are warning messages. |

## Hosts and Port Examples

The default port configured for Solr during the install process is 8983. The samples, URLs and screenshots in this guide may show different ports, because the port number that Solr uses is configurable. If you have not customized your installation of Solr, 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 Managing Solr.

Similarly, URL examples use 'localhost' throughout; if you are accessing Solr from a location remote to the server hosting Solr, replace 'localhost' with the proper domain or IP where Solr is running.

## Paths

Path information is given relative to `solr.home`, which is the location under the main Solr installation where Solr's collections and their `conf` and `data` directories are stored. In the default Solr package, `solr.home` is `example/solr`, which is itself relative to where you unpackaged the application; if you have moved this location for your servlet container or for another reason, the path to `solr.home` may be different than the default.

# Getting Started

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 ten times 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.

This section helps you get Solr up and running quickly, and introduces you to the basic Solr architecture and features. It covers the following topics:

Installing Solr: A walkthrough of the Solr installation process.

Running Solr: An introduction to running Solr. Includes information on starting up the servers, adding documents, and running queries.

A Quick Overview: A high-level overview of how Solr works.

A Step Closer: An introduction to Solr's home directory and configuration options.

# Installing Solr

This section describes how to install Solr. You can install Solr 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 section should work for any platform, with a few exceptions for Windows as noted.

### Got Java?

You will need the Java Runtime Environment (JRE) version 1.7 or higher. At a command line, check your Java version like this:

```
$ java -version
java version "1.7.0_55"
Java(TM) SE Runtime Environment (build 1.7.0_55-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.55-b03, mixed mode)
```

The output will vary, but you need to make sure you have version 1.7 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 Oracle at http://www.oracle.com/technetwork/java/javase/downloads/index.html.

### Installing Solr

Solr is available from the Solr website at http://lucene.apache.org/solr/.

For Linux/Unix/OSX systems, download the `.tgz` file. For Microsoft Windows systems, download the `.zip` file.

Solr runs inside a Java servlet container such as Tomcat, Jetty, or Resin. The Solr distribution includes a working demonstration server in the `Example` directory that runs in Jetty. You can use the example server as a template for your own installation, whether or not you are using Jetty as your servlet container. For more information about the demonstration server, see the Solr Tutorial.

> ⚠ Solr ships with a working Jetty server, with optimized settings for Solr, inside the `example` directory. It is recommended that you use the provided Jetty server for optimal performance. If you absolutely must use a different servlet container then continue to the next section on how to install Solr.

**To install Solr**

1. Unpack the Solr distribution to your desired location.
2. Stop your Java servlet container.
3. Copy the `solr.war` file from the Solr distribution to the `webapps` directory of your servlet container. Do not change the name of this file: it must be named `solr.war`.
4. Copy the Solr Home directory `solr-4.x.0/example/solr/` from the distribution to your desired Solr Home location.
5. Start your servlet container, passing to it the location of your Solr Home in one of these ways:
    - Set the Java system property `solr.solr.home` to your Solr Home. (for example, using the example jetty setup: `java -Dsolr.solr.home=/some/dir -jar start.jar`).
    - Configure the servlet container so that a JNDI lookup of `java:comp/env/solr/home` by the Solr webapp will point to your Solr Home.
    - Start the servlet container in the directory containing `./solr`: the default Solr Home is `solr` under the JVM's current working directory (`$CWD/solr`).

To confirm your installation, go to the Solr Admin page at `http://localhost:8983/solr/` . Note that your servlet container may have started on a different port: check the documentation for your servlet container to troubleshoot that issue. Also note that if that port is already in use, Solr will not start. In that case, shut down the servlet container running on that port, or change your Solr port.

For more information about installing and running Solr on different Java servlet containers, see the SolrInstall page on the Solr Wiki.

### Related Topics

- SolrInstall

## Running Solr

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

### Start the Server

If you didn't start Solr after installing it, you can start it by running `bin/solr` from the Solr directory.

```
$ bin/solr -f
```

If you are running Windows, you can start the Web server by running `bin\solr.cmd` instead.

```
C:\Applications\Solr\bin\solr.cmd -f
```

This will start Solr in the foreground, listening on port 8983. The `bin/solr` and `bin\solr.cmd` scripts allow you to customize how you start Solr. Let's work through a few examples of using the `bin/solr` script (if you're running

Solr on Windows, the `bin\solr.cmd` works the same as what is shown in the examples below):

### Solr Script Options

The `bin/solr` script has several options.

#### *Script Help*

To see how to use the `bin/solr` script, execute:

```
$ bin/solr -help
```

For specific usage instructions for the **start** command, do:

```
 $ bin/solr start -help
```

#### *Start Solr in the Background*

As you saw above, the `-f` flag will start Solr running in the foreground. Since Solr is a server, it is more common to run it in the background, especially on Unix/Linux. To start Solr running in the background, simply do:

```
 $ bin/solr start
```

When you start Solr in the background, the script will wait to make sure Solr starts correctly before returning to the command line prompt.

#### *Start Solr with a Different Port*

To change the port Solr listens on, you can use the `-p` parameter, such as:

```
 $ bin/solr start -p 8984
```

#### *Stop Solr*

When running Solr in the foreground (using -f), then you can stop it using `Ctrl-c`. However, when running in the background, you should use the **stop** command, such as:

```
 $ bin/solr stop
```

#### *Start Solr with a Specific Example Configuration*

Solr also provides a number of useful examples to help you learn about key features. You can launch the examples using the -e flag. For instance, to launch the Data Import Handler example, you would do:

```
 $ bin/solr -e dih
```

Currently, the available examples you can run are: default, dih, schemaless, and cloud.

#### *Check if Solr is Running*

If you're not sure if Solr is running locally, you can send the info flag (`-i`):

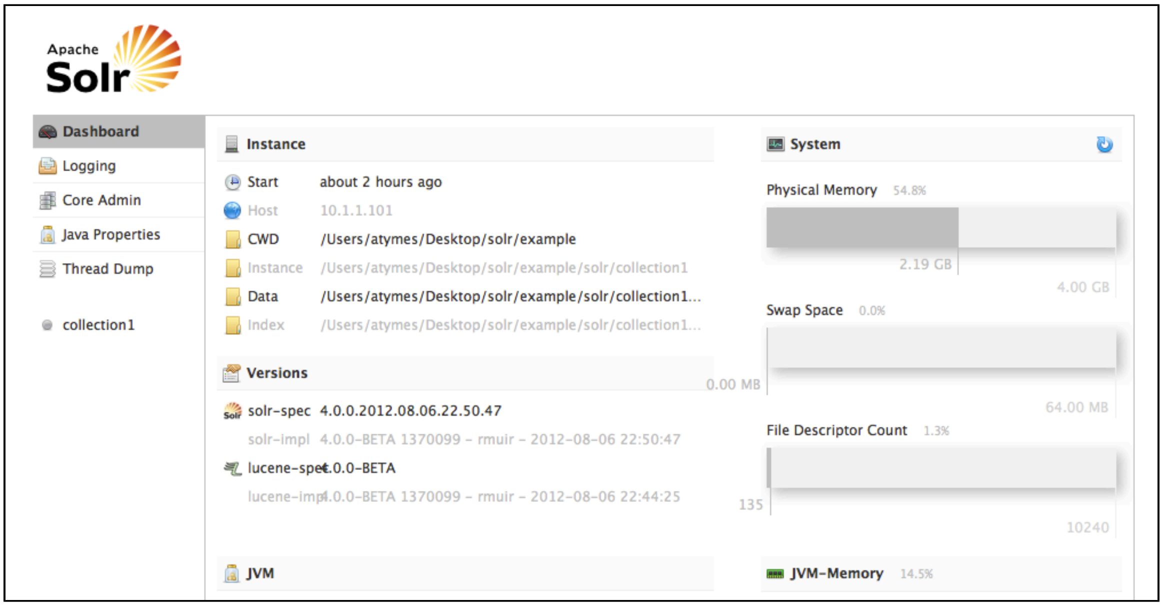
```
$ bin/solr -i
```

This will search for running Solr instances on your computer and then gather basic information about them, such as the version and memory usage.

For more information on starting Solr in cloud mode, see: Getting Started with SolrCloud.

That's it! Solr is running. If you need convincing, use a Web browser to see the Admin Console.

`http://localhost:8983/solr/`



Start about 2 hours ago
Host 10.1.1.101
CWD /Users/atymes/Desktop/solr/example
Instance /Users/atymes/Desktop/solr/example/solr/collection1
Data /Users/atymes/Desktop/solr/example/solr/collection1...
Index /Users/atymes/Desktop/solr/example/solr/collection1...

solr-spec 4.0.0.2012.08.06.22.50.47
solr-impl 4.0.0-BETA 1370099 - rmuir - 2012-08-06 22:50:47
lucene-spec 4.0.0-BETA
lucene-impl 4.0.0-BETA 1370099 - rmuir - 2012-08-06 22:44:25

*The Solr Admin interface.*

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

### Add Documents

Solr is built to find documents that match queries. Solr's schema provides an idea of how content is structured (more on the schema later), but without documents there is nothing to find. Solr needs input before it can do anything.

You may want to add a few sample documents before trying to index your own content. The Solr installation comes with example documents located in the `example/exampledocs` directory of your installation.

In the `exampledocs` directory is the SimplePostTool, a Java-based command line tool, `post.jar`, which can be used to index the documents. Do not worry too much about the details for now. The Indexing and Basic Data Operations section 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
```

The SimplePostTool 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 arguments, as raw command line `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..
Time spent: 0:00:00.633
$
```

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

### Ask Questions

Now that you have 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.

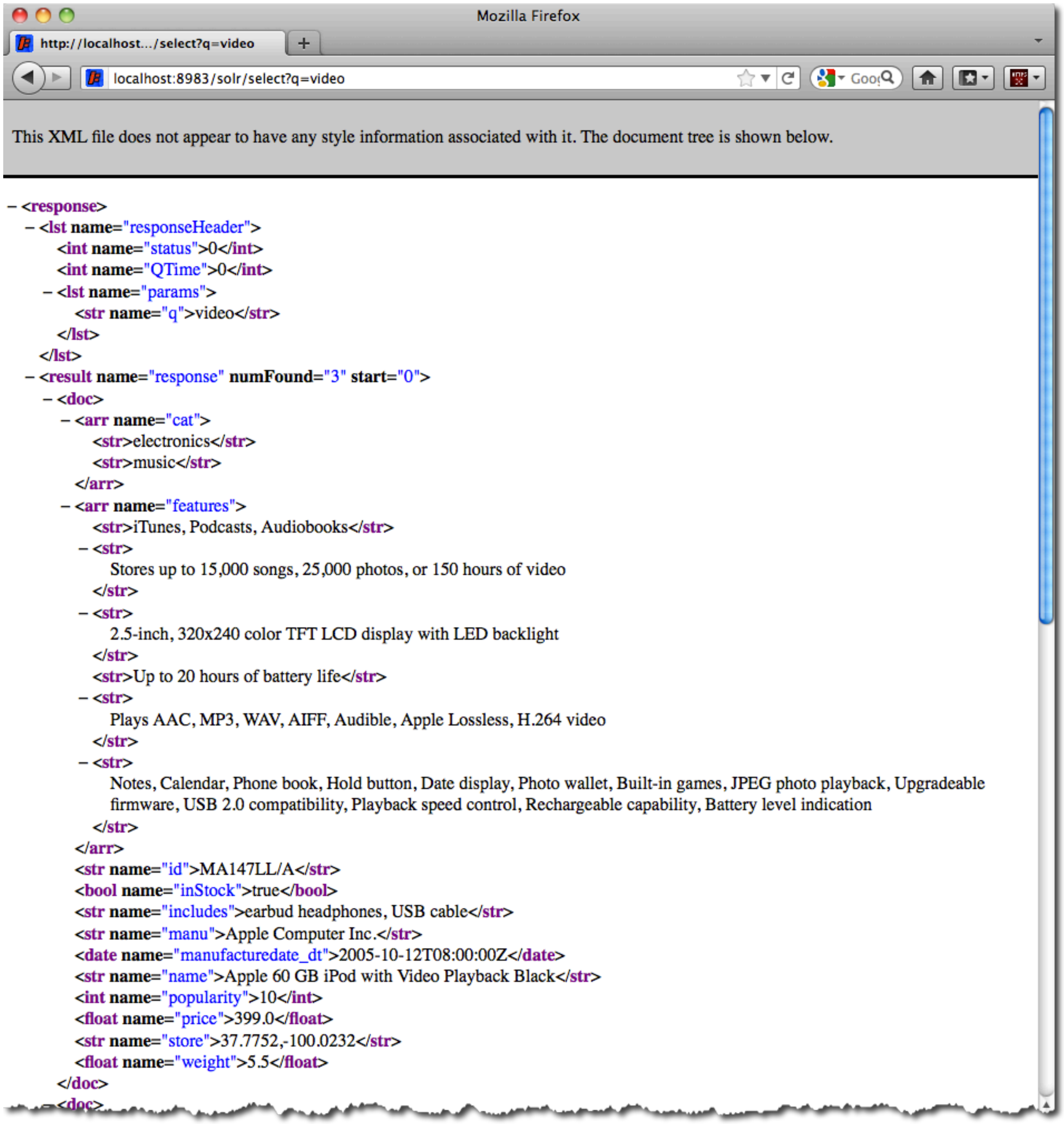
For example, the following query searches all document fields for "video":

`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 main 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 shot shows the result of a query (the next example, actually) as viewed in Mozilla Firefox. The top-level response contains a `lst` named `responseHeader` a nd a result named response. Inside result, you can see the three docs that represent the search results.


Mozilla Firefox
http://localhost.../select?q=video
localhost:8983/solr/select?q=video
Goo

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
- <response>
  - <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
    - <lst name="params">
        <str name="q">video</str>
      </lst>
    </lst>
  - <result name="response" numFound="3" start="0">
    - <doc>
      - <arr name="cat">
          <str>electronics</str>
          <str>music</str>
        </arr>
      - <arr name="features">
          <str>iTunes, Podcasts, Audiobooks</str>
        - <str>
            Stores up to 15,000 songs, 25,000 photos, or 150 hours of video
          </str>
        - <str>
            2.5-inch, 320x240 color TFT LCD display with LED backlight
          </str>
          <str>Up to 20 hours of battery life</str>
        - <str>
            Plays AAC, MP3, WAV, AIFF, Audible, Apple Lossless, H.264 video
          </str>
        - <str>
            Notes, Calendar, Phone book, Hold button, Date display, Photo wallet, Built-in games, JPEG photo playback, Upgradeable
            firmware, USB 2.0 compatibility, Playback speed control, Rechargeable capability, Battery level indication
          </str>
        </arr>
        <str name="id">MA147LL/A</str>
        <bool name="inStock">true</bool>
        <str name="includes">earbud headphones, USB cable</str>
        <str name="manu">Apple Computer Inc.</str>
        <date name="manufacturedate_dt">2005-10-12T08:00:00Z</date>
        <str name="name">Apple 60 GB iPod with Video Playback Black</str>
        <int name="popularity">10</int>
        <float name="price">399.0</float>
        <str name="store">37.7752,-100.0232</str>
        <float name="weight">5.5</float>
      </doc>
    - <doc>
```

*An XML response to a query.*

Once you have mastered the basic idea of a query, it is 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 do not 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%20TO%20400]&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%20TO%20400]&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**

```
<response>
<lst name="responseHeader">
...
</lst>
<result name="response" numFound="9" start="0">
  <doc>
    <str name="id">SOLR1000</str>
    <str name="name">Solr, the Enterprise Search Server</str>
    <float name="price">0.0</float></doc>
...
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="cat">
      <int name="electronics">6</int>
      <int name="memory">3</int>
      <int name="search">2</int>
      <int name="software">2</int>
      <int name="camera">1</int>
      <int name="copier">1</int>
      <int name="multifunction printer">1</int>
      <int name="music">1</int>
      <int name="printer">1</int>
      <int name="scanner">1</int>
      <int name="connector">0</int>
      <int name="currency">0</int>
      <int name="graphics card">0</int>
      <int name="hard drive">0</int>
      <int name="monitor">0</int>
    </lst>
  </lst>
  <lst name="facet_dates"/>
  <lst name="facet_ranges"/>
</lst>
</response>
```

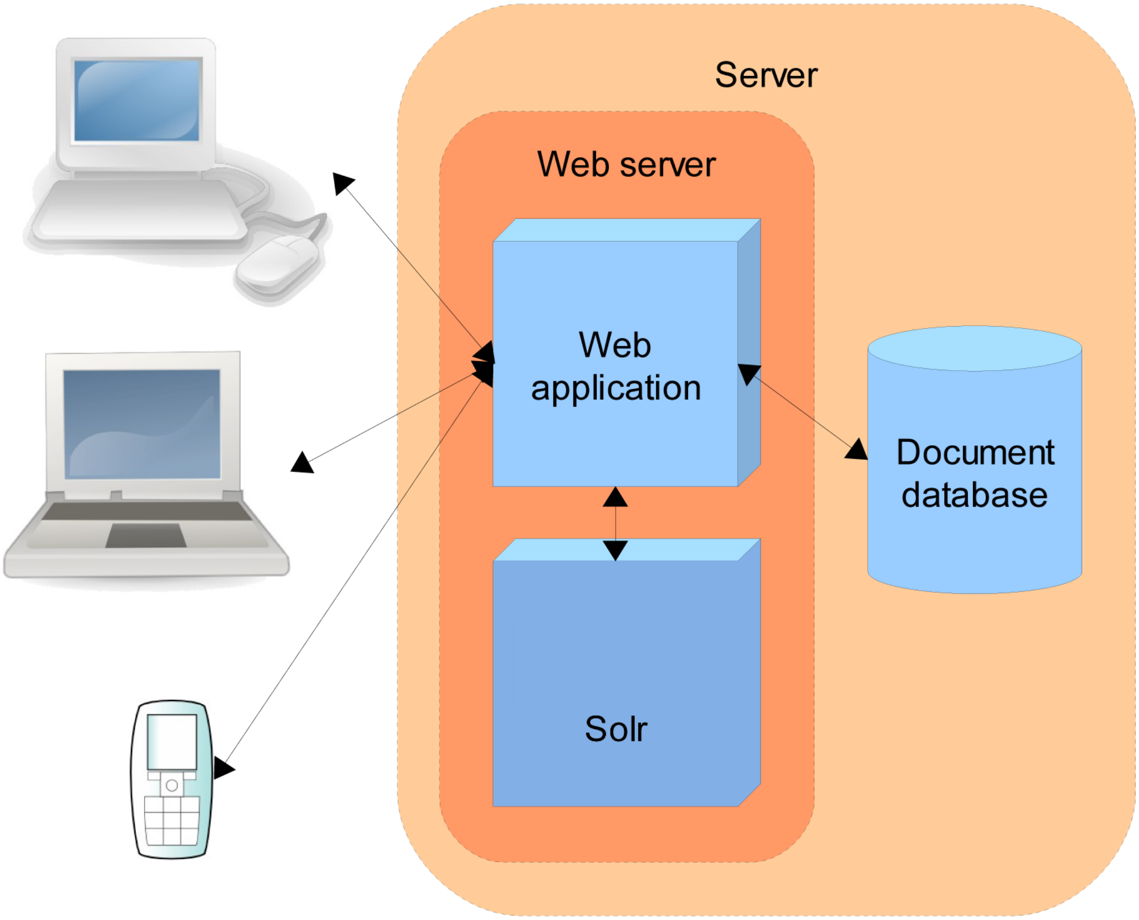
The facet information shows how many of the query results have each possible value of the `cat` field. You could easily use this information to provide users with a quick way to narrow their query results. You can filter results by adding one or more filter queries to the Solr request. Here is a request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

```
http://localhost:8983/solr/select?q=price:[0%20TO%20400]&fl=id,name,price&facet=true&
facet.field=cat&fq=cat:software
```

## A Quick Overview

Having had some fun with Solr, you will now learn about all the cool things it can do.

Here is a typical configuration:



*In the scenario above, Solr runs alongside another application in a Web server. 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:

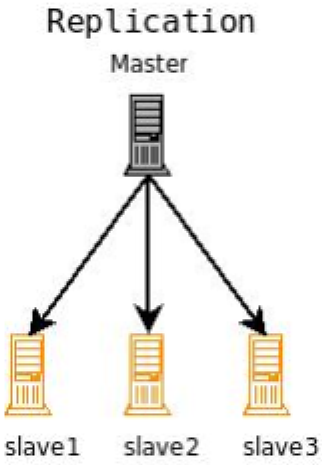
1. Define a *schema*. The schema tells Solr about the contents of documents it will be indexing. In the online store example, the schema would define fields for the product name, description, price, manufacturer, and so on. Solr's schema is powerful and flexible and allows you to tailor Solr's behavior to your application. See Documents, Fields, and Schema Design for all the details.
2. Deploy Solr to your application server.
3. Feed Solr the document for which your users will search.
4. 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 it could also be JSON, CSV, 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 Client APIs for details on client APIs.

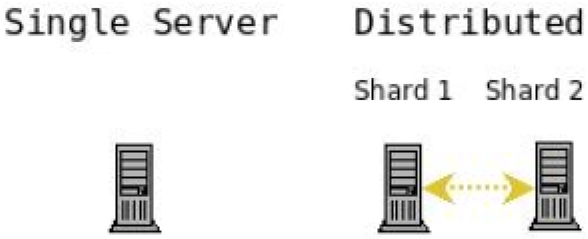
Solr is based on the Apache Lucene project, a high-performance, full-featured search engine. Solr offers support for the simplest keyword searching through to complex queries on multiple fields and faceted search results. Searching has more information about searching and queries.

If Solr's capabilities are not impressive enough, its ability to handle very high-volume applications should do the trick.

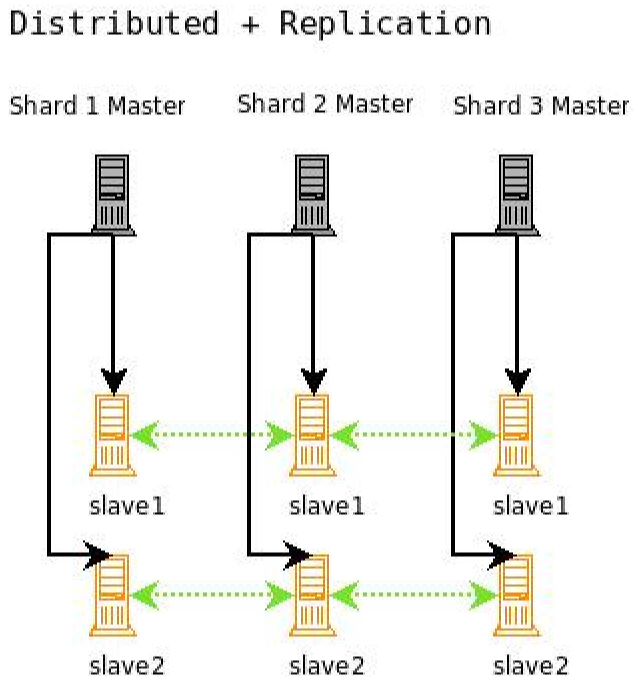
A relatively common scenario is that you have so many queries that the server is unable to respond fast enough to each one. In this case, you can make copies of an index. This is called replication. Then you can distribute incoming queries among the copies in any way you see fit. A round-robin mechanism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



Another useful technique is sharding. If you have so many documents that you simply cannot 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 have huge numbers of documents and users, you might need to combine the techniques of sharding and replication. In this case, Solr's new SolrCloud functionality may be more effective for your needs. SolrCloud includes a number of features to simplify the process of distributing the index and the queries, and manage the resulting nodes.



For full details on sharding and replication, see Legacy Scaling and Distribution. We've split the SolrCloud information into it's own section, called SolrCloud.

Best of all, this talk about high-volume applications is not just hypothetical: some of the famous Internet sites that use Solr today are Macy's, EBay, and Zappo's.

For more information, take a look at https://wiki.apache.org/solr/PublicServers.

## 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>/
  solr.xml
  conf/
    solrconfig.xml
    schema.xml
  data/
```

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

`solr.xml` specifies configuration options for your Solr core, and also allows you to configure multiple cores. For more information on `solr.xml` see The Well-Configured Solr Instance.

`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 The Well-Configured Solr Instance.

`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 Documents, Fields, and Schema Design.

# Upgrading Solr

If you are already using Solr 4.9, Solr 4.10 should not present any major problems. However, you should review the `CHANGES.txt` file found in your Solr package for changes and updates that may effect your existing implementation.

## Upgrading from 4.9.x

- In Solr 3.6, all primitive field types were changed to omit norms by default when the schema version is 1.5 or greater (SOLR-3140), but `TrieDateField`'s default was mistakenly not changed.  As of Solr 4.10, `TrieDateField` omits norms by default (see SOLR-6211).
- Creating a `SolrCore` via `CoreContainer.create()` no longer requires an additional call to `CoreContainer.register()` to make it available to clients (see SOLR-6170).
- `CoreContainer.remove()` has been removed.  You should now use `CoreContainer.unload()` to delete a `SolrCore` (see SOLR-6232).
- `solr.xml` parsing has been improved to better account for the expected data types of various options.  As part of this fix, additional error checking has also been added to provide errors in the event of duplicated options, or unknown option names that may indicate a typo.  Users who have modified their solr.xml in the past and now upgrade may get errors on startup if they have typos or unexpected options specified in their `solr.xml` file.  (See SOLR-5746 for more information.)

## Upgrading from Older Versions of Solr

This is a summary of some of the key issues related to upgrading in previous versions of Solr. Users upgrading from older versions are strongly encouraged to consult `CHANGES.txt` for the details of all changes since the version they are upgrading from.

- In Solr 4.9, Support for DiskDocValuesFormat (i.e., fieldTypes configured with `docValuesFormat="Disk"`) was removed due to poor performance. If you have existing fieldTypes using DiskDocValuesFormat please modify your schema.xml to remove the 'docValuesFormat' attribute, and optimize your index to rewrite it into the default codec prior to upgrading to 4.9 or later. See LUCENE-5761 for more details.
- Begining with Solr 4.8, Java 7 or greater is required. When using Oracle Java 7 or OpenJDK 7, be sure to not use the GA build 147 or update versions u40, u45 and u51! We recommend using u55 or later.  An overview of known JVM bugs can be found on http://wiki.apache.org/lucene-java/JavaBugs
- Prior to Solr 4.8, terms that exceeded Lucene's `MAX_TERM_LENGTH` were silently ignored when indexing documents. Begining with Solr 4.8, a document an error will be generated when attempting to index a document with a term that is too large. If you wish to continue to have large terms ignored, use `solr.LengthFilterFactory` in all of your Analyzers. See LUCENE-5472 for more details.
- The `<fields>` and `<types>` tags in `schema.xml` was deprecated in Solr 4.8.  There is no longer any reason to keep them in the schema file, they may be safely removed. This allows intermixing of `<fieldType>`, `<field>` and `<copyField>` definitions if desired. Currently, these tags are supported so either style may be implemented. They may be deprecated formally in 5.0.  See `SOLR-5228` for more details.
- In Solr 4.7, due to a bug in previous versions the default value of the `discountOverlap` property of `DefaultSimilarity` was not being set appropriately if you were using the implicit `DefaultSimilarityFactory` instead of explicitly configuring it.  To preserve consistent behavior for people who upgrade, the implicit behavior is now contingent on the `<luceneMatchVersion/>` -- `discountOverlap=false` for 4.6 and below, `discountOverlap=true` for 4.7 and above.  See SOLR-5561 for more information.
- In Solr 4.6, The "file" attribute of infoStream in `solrconfig.xml` was removed. Control this via your logging

configuration (org.apache.solr.update.LoggingInfoStream) instead.

- In Solr 4.5, XML configuration parsing was made more strict about situations where a single setting is allowed but multiple values are found. Configuration parsing now fails with an error in situations like this. Also, `schema.xml` parsing was also made more strict: "default" or "required" options specified on `<dynamicField/>` declarations will cause an init error. You can safely remove these attributes.
- In Solr 4.5, `CloudSolrServer` can now use multiple threads to add documents by default. This is a small change in runtime semantics when using the bulk add method - you will still end up with the same exception on a failure, but some documents beyond the one that failed may have made it in. To get the old, single threaded behavior, set parallel updates to false on the `CloudSolrServer` instance.
- Beginning with 4.4, the use of the Compound File Format is determined by IndexWriter configuration, and not the Merge Policy. If you have explicitly configured a `<mergePolicy>` with the `setUseCompoundFile` configuration option, you should change this to use the `useCompoundFile` configuration option directly in the `<indexConfig>` block. Specifying `setUseCompoundFile` on the Merge Policy will no longer work in Solr 5.0.
- In Solr 4.4, `ByteField` and `ShortField` were deprecated, and will be removed in 5.0. Please switch to using `TrieIntField`
- The pre-4.3.0 `solr.xml` "legacy" mode and format will no longer be supported in Solr 5.0. Users are encouraged to migrate from "legacy" to "discovery" `solr.xml` configurations, see Solr Cores and solr.xml.
- As of Solr 4.3 the slf4j/logging jars are no longer included in the Solr webapp to allow for more flexibility in logging.
- Minor changes were made to the Schema API response format in Solr 4.3
- In Solr 4.1 the method Solr uses to identify node names for SolrCloud was changed. If you are using SolrCloud and upgrading from Solr 4.0, you may have issues with unknown or lost nodes. If this occurs, you can manually set the `host` parameter either in `solr.xml` or as a system variable. More information can be found in the section on SolrCloud.
- If you are upgrading from Solr 3.x, you should familiarize yourself with the Major Changes from Solr 3 to Solr 4.

# Using the Solr Administration User Interface

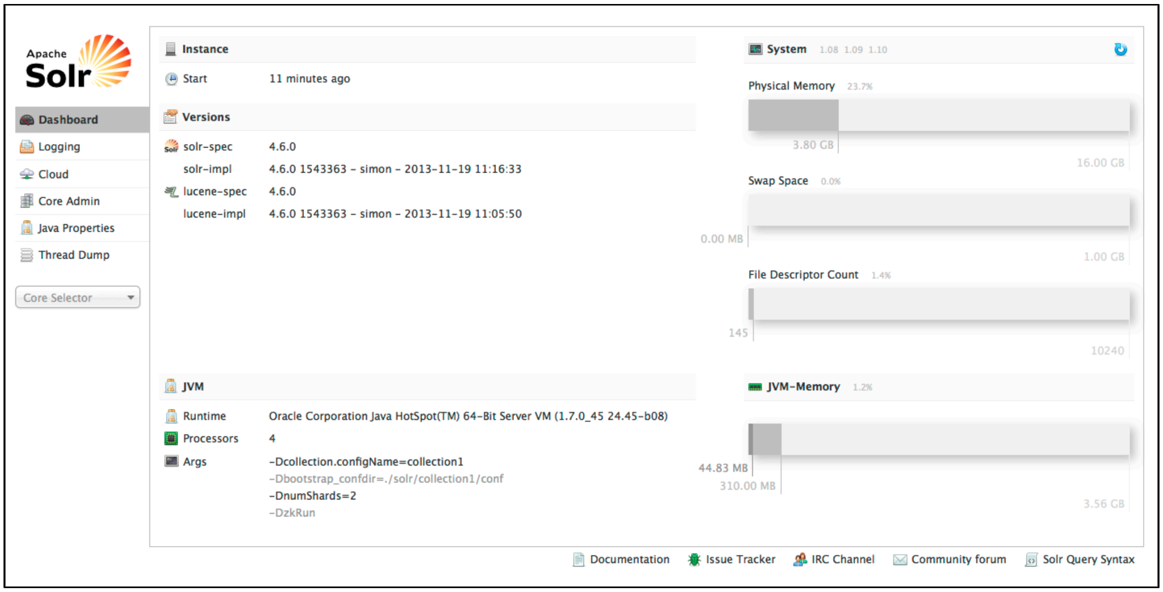
This section discusses the Solr Administration User Interface ("Admin UI").

The Overview of the Solr Admin UI explains how the features of the user interface that are new with Solr 4, what's on the initial Admin UI page, and how to configure the interface. In addition, there are pages describing each screen of the Admin UI:

- **Getting Assistance** shows you how to get more information about the UI.
- **Logging** explains the various logging levels available and how to invoke them.
- **Cloud Screens** display information about nodes when running in SolrCloud mode.
- **Core Admin** explains how to get management information about each core.
- **Java Properties** shows the Java information about each core.
- **Thread Dump** lets you see detailed information about each thread, along with state information.
- **Core-Specific Tools** is a section explaining additional screens available for each named core.
    - Analysis - lets you analyze the data found in specific fields.
    - Dataimport - shows you information about the current status of the Data Import Handler.
    - Documents - provides a simple form allowing you to execute various Solr indexing commands directly from the browser.
    - Files - shows the current core configuration files such as `solrconfig.xml` and `schema.xml`.
    - Ping - lets you ping a named core and determine whether the core is active.
    - Plugins/Stats - shows statistics for plugins and other installed components.
    - Query - lets you submit a structured query about various elements of a core.
    - Replication - shows you the current replication status for the core, and lets you enable/disable replication.
    - Schema Browser - displays schema data in a browser window.

## Overview of the Solr Admin UI

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 and access online documentation and other help.



Start 11 minutes ago

solr-spec 4.6.0
solr-impl 4.6.0 1543363 - simon - 2013-11-19 11:16:33
lucene-spec 4.6.0
lucene-impl 4.6.0 1543363 - simon - 2013-11-19 11:05:50

Runtime Oracle Corporation Java HotSpot(TM) 64-Bit Server VM (1.7.0_45 24.45-b08)
Processors 4
Args -Dcollection.configName=collection1
-Dbootstrap_confdir=./solr/collection1/conf
-DnumShards=2
-DzkRun

With Solr 4, the Solr Admin has been completely redesigned. The redesign was completed with these benefits in mind:

- load pages quicker
- access and control functionality from the Dashboard
- re-use the same servlets that access Solr-related data from an external interface, and
- ignore any differences between working with one or multiple cores.

Accessing the URL `http://hostname:8983/solr/` (if running Jetty on the default port of 8983), will show the main dashboard, which is divided into two parts.

A left-side of the screen is a menu under the Solr logo that provides the navigation through the screens of the UI. The first set of links are for system-level information and configuration and provide access to Logging, Core Admin and Java Properties, among other things. At the end of this information is a list of Solr cores configured for this instance. Clicking on a core name shows a secondary menu of information and configuration options for the core specifically. Items in this list include the Schema, Config, Plugins, and an ability to perform Queries on indexed data.

The center of the screen shows the detail of the option selected. This may include a sub-navigation for the option or text or graphical representation of the requested data. See the sections in this guide for each screen for more details.

### Configuring the Admin UI in `solrconfig.xml`

You can configure the Solr Admin UI by editing the file `solrconfig.xml`.

The `<admin>` block in the `solrconfig.xml` file determines the default query to be displayed in the Query section of the core-specific pages. The default is `*:*`, which is to find all documents. In this example, we have changed the default to the term `solr`.

```
<admin>
    <defaultQuery>solr</defaultQuery>
</admin>
```

### Related Topics

- Configuring solrconfig.xml

## Getting Assistance

At the bottom of each screen of the Admin UI is a set of links that can be used to get more assistance with configuring and using Solr.

*Assistance icons*

These icons include the following links.

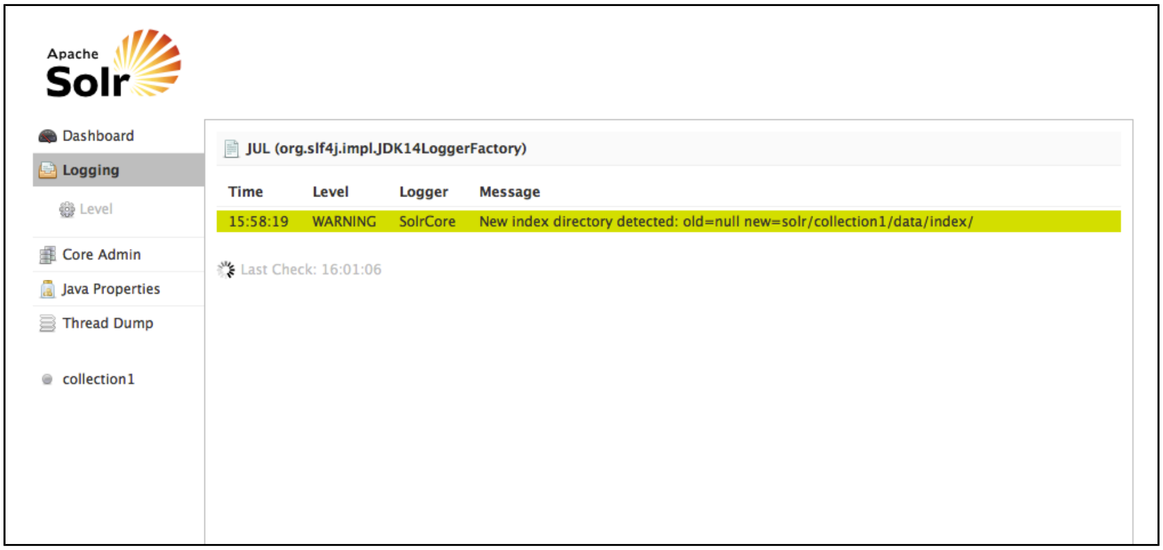
| Link | Description |
|---|---|
| Documentation | Navigates to the Apache Solr documentation hosted on http://lucene.apache.org/solr/. |
| Issue Tracker | Navigates to the JIRA issue tracking server for the Apache Solr project. This server resides at http://issues.apache.org/jira/browse/SOLR. |
| IRC Channel | Connects you to the web interface for Solr's IRC channel. This channel is found on `irc.freenode.net`, Port 7000, `#solr` channel. |
| Community forum | Connects you to the Solr community forum, which at the current time is a set of mailing lists and their archives. |
| Solr Query Syntax | Navigates to the Apache Wiki page describing the Solr query syntax: http://wiki.apache.org/solr/SolrQuerySyntax. |

These links cannot be modified without editing the `admin.html` in the `solr.war` that contains the Admin UI files.

## Logging

The Logging page shows messages from Solr's log files.

When you click the link for "Logging", a page similar to the one below will be displayed:



JUL (org.slf4j.impl.JDK14LoggerFactory)

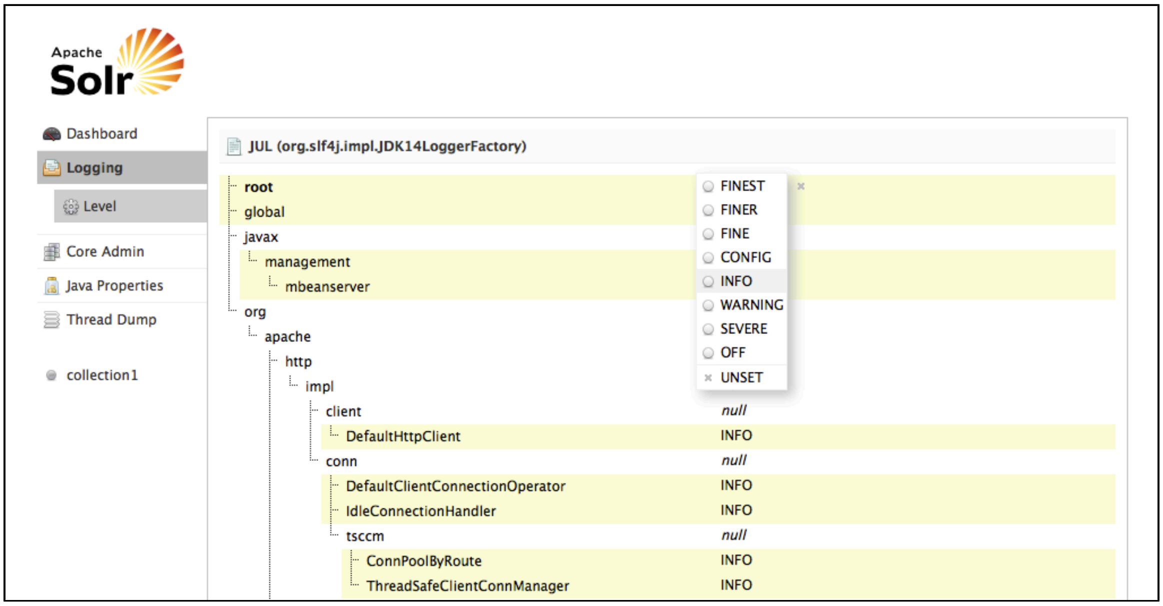
| Time | Level | Logger | Message |
|---|---|---|---|
| 15:58:19 | WARNING | SolrCore | New index directory detected: old=null new=solr/collection1/data/index/ |

Last Check: 16:01:06

*The Main Logging Screen*

While this example shows logged messages for only one core, if you have multiple cores in a single instance, they will each be listed, with the level for each.

### Selecting a Logging Level



JUL (org.slf4j.impl.JDK14LoggerFactory)

```
root
global
javax
  management
    mbeanserver
org
  apache
    http
      impl
        client                           null
          DefaultHttpClient              INFO
        conn                             null
          DefaultClientConnectionOperator INFO
          IdleConnectionHandler          INFO
          tsccm                          null
            ConnPoolByRoute              INFO
            ThreadSafeClientConnManager  INFO
```

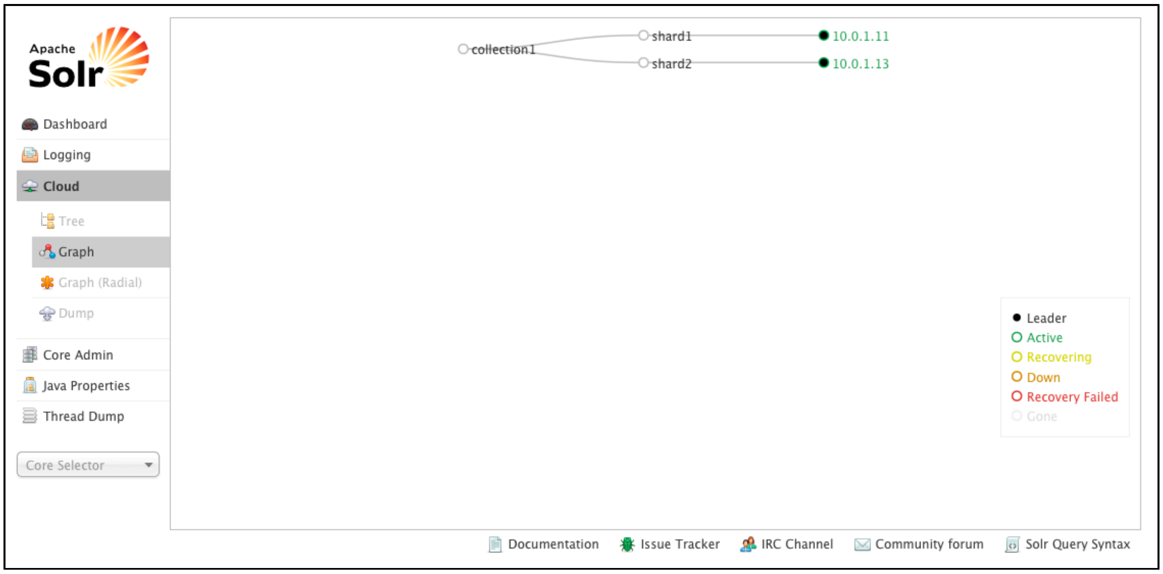
When you select the **Level** link on the left, you see the hierarchy of classpaths and classnames for your instance. A row highlighted in yellow indicates that the class has logging capabilities. Click on a highlighted row, and a menu will appear to allow you to change the log level for that class. Characters in boldface indicate that the class will not be affected by level changes to root.

For an explanation of the various logging levels, see Configuring Logging.

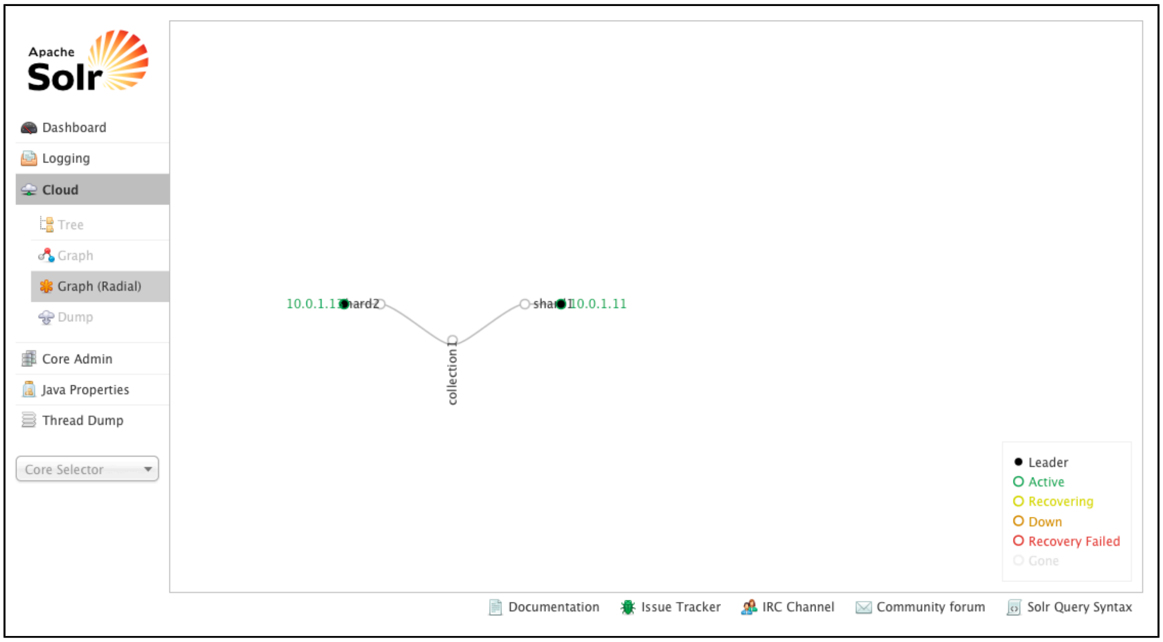
## Cloud Screens

When running in SolrCloud mode, an option will appear in the Admin UI between Logging and Core Admin for Cloud. It's not possible at the current time to manage the nodes of the SolrCloud cluster, but you can view them and open the Solr Admin UI on each node to view the status and statistics for the node and each core on each node.

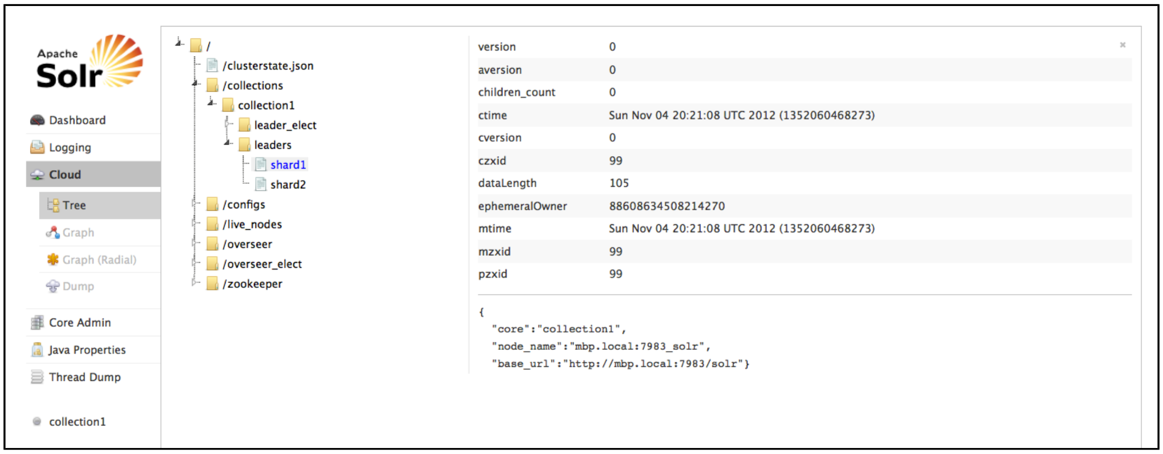
Click on the Cloud option in the left-hand navigation, and a small sub-menu appears with options called "Tree", "Graph", "Graph (Radial)" and "Dump". The default view (which is "Tree") shows a graph of each core and the addresses of each node. This example shows a very simple two-node cluster with a single core:



The "Graph (Radial)" option provides a different visual view of each node. Using the same simple two-node cluster, the radial graph view looks like:



The "Tree" option shows a directory structure of the files in ZooKeeper, including `clusterstate.json`, configuration files, and other status and information files. In this example, we show the leader definition files for the core named "collection1":



```
/
  /clusterstate.json
  /collections
    collection1
      leader_elect
      leaders
        shard1
        shard2
  /configs
  /live_nodes
  /overseer
  /overseer_elect
  /zookeeper
```

| | |
|---|---|
| version | 0 |
| aversion | 0 |
| children_count | 0 |
| ctime | Sun Nov 04 20:21:08 UTC 2012 (1352060468273) |
| cversion | 0 |
| czxid | 99 |
| dataLength | 105 |
| ephemeralOwner | 88608634508214270 |
| mtime | Sun Nov 04 20:21:08 UTC 2012 (1352060468273) |
| mzxid | 99 |
| pzxid | 99 |

```
{
  "core":"collection1",
  "node_name":"mbp.local:7983_solr",
  "base_url":"http://mbp.local:7983/solr"}
```

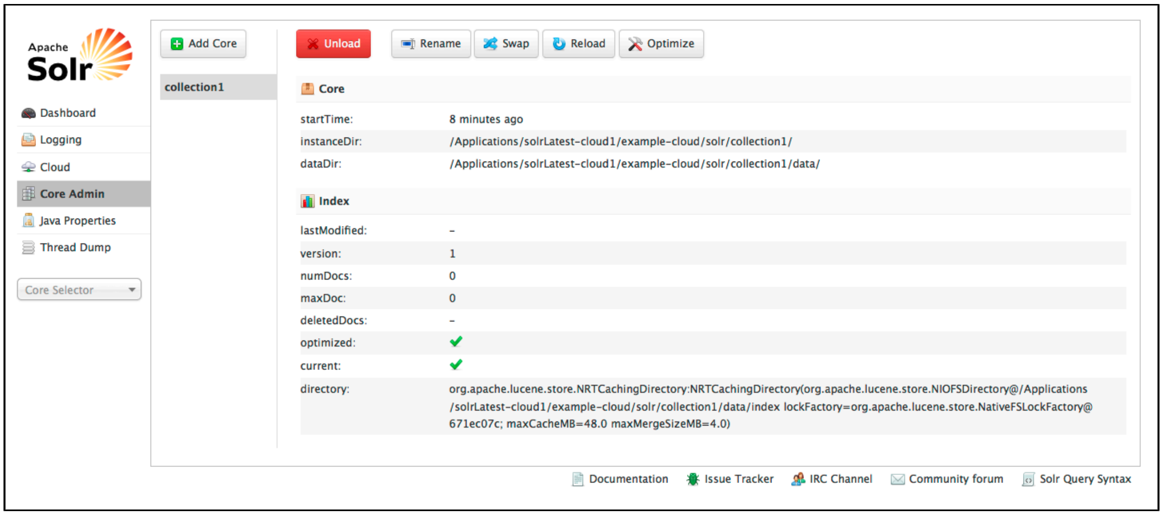
The final option is "Dump", which allows you to download an XML file with all the ZooKeeper configuration files.

## Core Admin

The Core Admin screen lets you manage your cores.

The buttons at the top of the screen let you add a new core, unload the core displayed, rename the currently displayed core, swap the existing core with one that you specify in a drop-down box, reload the current core, and optimize the current core.

The main display and available actions correspond to the commands used with the CoreAdminHandler, but provide another way of working with your cores.



**Core**

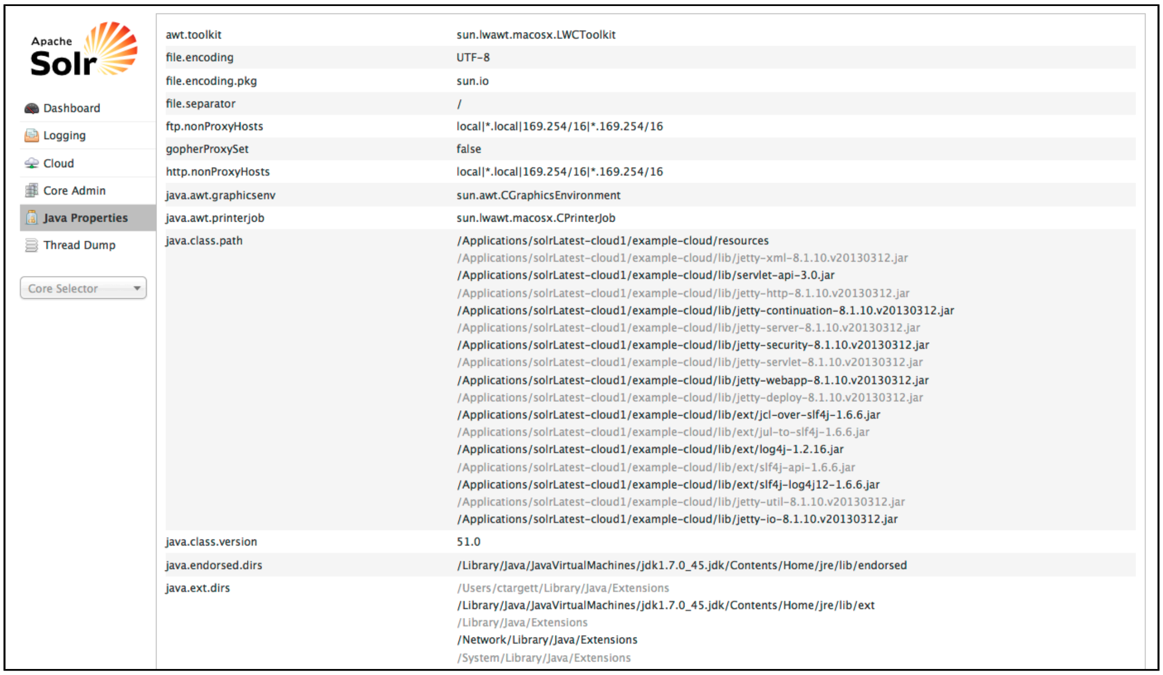
| | |
|---|---|
| startTime: | 8 minutes ago |
| instanceDir: | /Applications/solrLatest-cloud1/example-cloud/solr/collection1/ |
| dataDir: | /Applications/solrLatest-cloud1/example-cloud/solr/collection1/data/ |

**Index**

| | |
|---|---|
| lastModified: | - |
| version: | 1 |
| numDocs: | 0 |
| maxDoc: | 0 |
| deletedDocs: | - |
| optimized: | ✓ |
| current: | ✓ |
| directory: | org.apache.lucene.store.NRTCachingDirectory:NRTCachingDirectory(org.apache.lucene.store.NIOFSDirectory@/Applications /solrLatest-cloud1/example-cloud/solr/collection1/data/index lockFactory=org.apache.lucene.store.NativeFSLockFactory@ 671ec07c; maxCacheMB=48.0 maxMergeSizeMB=4.0) |

## Java Properties

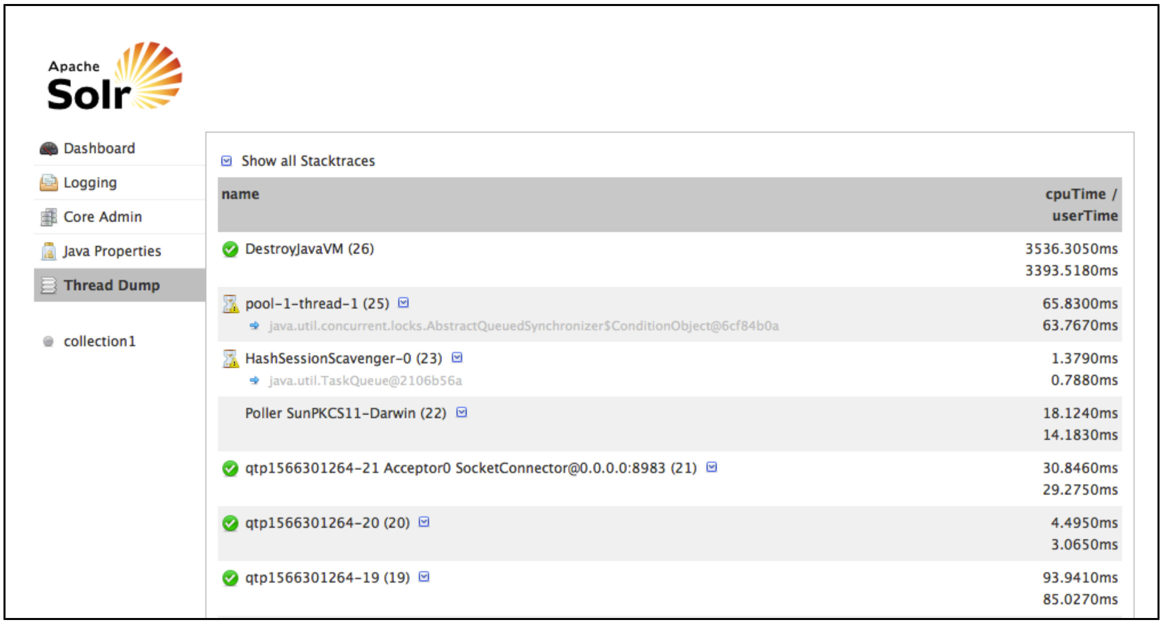
The Java Properties screen provides easy access to one of the most essential components of a top-performing Solr systems With the Java Properties screen, you can see all the properties of the JVM running Solr, including the class paths, file encodings, JVM memory settings, operating system, and more.



| | |
|---|---|
| awt.toolkit | sun.lwawt.macosx.LWCToolkit |
| file.encoding | UTF-8 |
| file.encoding.pkg | sun.io |
| file.separator | / |
| ftp.nonProxyHosts | local\|*.local\|169.254/16\|*.169.254/16 |
| gopherProxySet | false |
| http.nonProxyHosts | local\|*.local\|169.254/16\|*.169.254/16 |
| java.awt.graphicsenv | sun.awt.CGraphicsEnvironment |
| java.awt.printerjob | sun.lwawt.macosx.CPrinterJob |
| java.class.path | /Applications/solrLatest-cloud1/example-cloud/resources<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-xml-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/servlet-api-3.0.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-http-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-continuation-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-server-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-security-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-servlet-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-webapp-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-deploy-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/ext/jcl-over-slf4j-1.6.6.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/ext/jul-to-slf4j-1.6.6.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/ext/log4j-1.2.16.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/ext/slf4j-api-1.6.6.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/ext/slf4j-log4j12-1.6.6.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-util-8.1.10.v20130312.jar<br>/Applications/solrLatest-cloud1/example-cloud/lib/jetty-io-8.1.10.v20130312.jar |
| java.class.version | 51.0 |
| java.endorsed.dirs | /Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home/jre/lib/endorsed |
| java.ext.dirs | /Users/ctargett/Library/Java/Extensions<br>/Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/Home/jre/lib/ext<br>/Library/Java/Extensions<br>/Network/Library/Java/Extensions<br>/System/Library/Java/Extensions |

## Thread Dump

The Thread Dump screen lets you inspect the currently active threads on your server. Each thread is listed and access to the stacktraces is available where applicable. Icons to the left indicate the state of the thread: for example, threads with a green check-mark in a green circle are in a "RUNNABLE" state. On the right of the thread name, a down-arrow means you can expand to see the stacktrace for that thread.

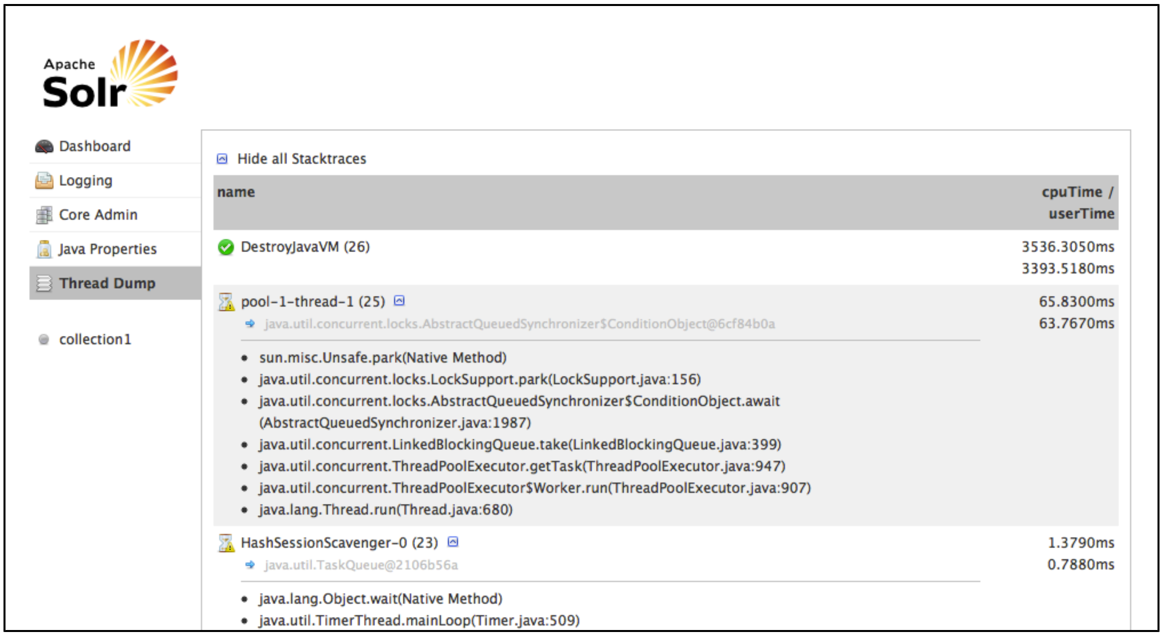


| name | cpuTime /<br>userTime |
|---|---|
| ✓ DestroyJavaVM (26) | 3536.3050ms<br>3393.5180ms |
| pool-1-thread-1 (25)<br>➔ java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@6cf84b0a | 65.8300ms<br>63.7670ms |
| HashSessionScavenger-0 (23)<br>➔ java.util.TaskQueue@2106b56a | 1.3790ms<br>0.7880ms |
| Poller SunPKCS11-Darwin (22) | 18.1240ms<br>14.1830ms |
| ✓ qtp1566301264-21 Acceptor0 SocketConnector@0.0.0.0:8983 (21) | 30.8460ms<br>29.2750ms |
| ✓ qtp1566301264-20 (20) | 4.4950ms<br>3.0650ms |
| ✓ qtp1566301264-19 (19) | 93.9410ms<br>85.0270ms |

When you move your cursor over a thread name, a box floats over the name with the state for that thread. Thread states can be:

| State | Meaning |
|---|---|
| NEW | A thread that has not yet started. |
| RUNNABLE | A thread executing in the Java virtual machine. |
| BLOCKED | A thread that is blocked waiting for a monitor lock. |
| WAITING | A thread that is waiting indefinitely for another thread to perform a particular action. |
| TIMED_WAITING | A thread that is waiting for another thread to perform an action for up to a specified waiting time. |
| TERMINATED | A thread that has exited. |

When you click on one of the threads that can be expanded, you'll see the stacktrace, as in the example below:



| name | cpuTime / userTime |
|---|---|
| DestroyJavaVM (26) | 3536.3050ms<br>3393.5180ms |
| pool-1-thread-1 (25)<br>java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@6cf84b0a | 65.8300ms<br>63.7670ms |
| HashSessionScavenger-0 (23)<br>java.util.TaskQueue@2106b56a | 1.3790ms<br>0.7880ms |

pool-1-thread-1 (25):

```
• sun.misc.Unsafe.park(Native Method)
• java.util.concurrent.locks.LockSupport.park(LockSupport.java:156)
• java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await
  (AbstractQueuedSynchronizer.java:1987)
• java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:399)
• java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:947)
• java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)
• java.lang.Thread.run(Thread.java:680)
```

HashSessionScavenger-0 (23):

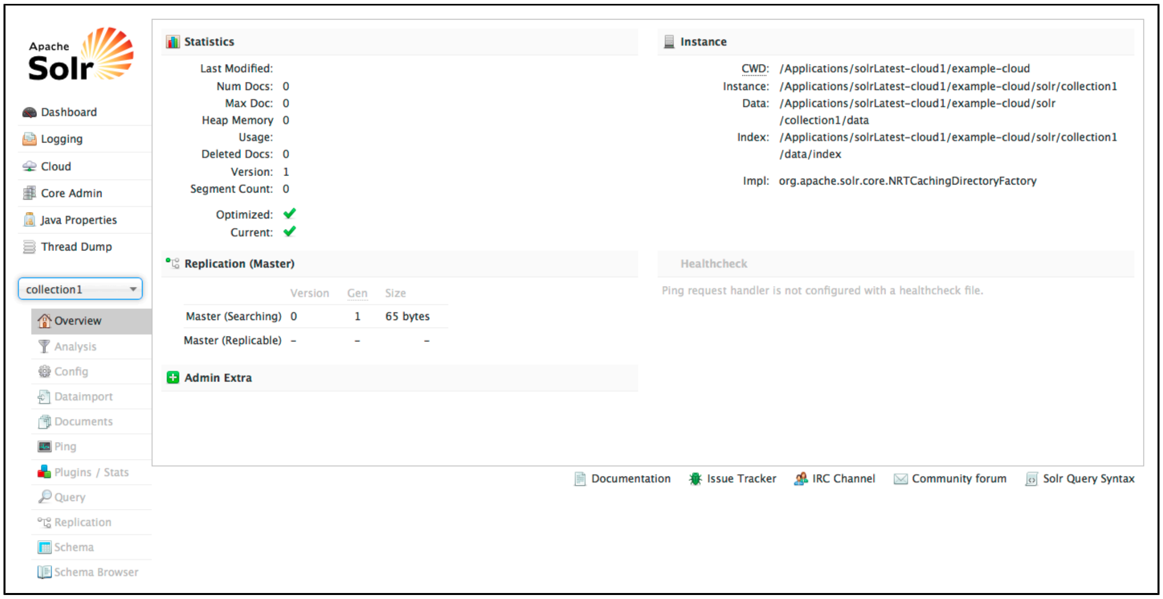
```
• java.lang.Object.wait(Native Method)
• java.util.TimerThread.mainLoop(Timer.java:509)
```

*Inspecting a thread*

You can also check the **Show all Stacktraces** button to automatically enable expansion for all threads.

## Core-Specific Tools

In the left-hand navigation bar, you will see a pull-down menu titled "Core Selector". Clicking on the menu will show a list of Solr cores, with a search box that can be used to find a specific core (handy if you have a lot of cores). When you select a core, such as **collection1** in the example, a secondary menu opens under the core name with the administration options available for that particular core.



Statistics

Last Modified:
Num Docs: 0
Max Doc: 0
Heap Memory Usage: 0
Deleted Docs: 0
Version: 1
Segment Count: 0
Optimized: ✓
Current: ✓

Replication (Master)

| | Version | Gen | Size |
|---|---|---|---|
| Master (Searching) | 0 | 1 | 65 bytes |
| Master (Replicable) | - | - | - |

Admin Extra

Instance

CWD: /Applications/solrLatest-cloud1/example-cloud
Instance: /Applications/solrLatest-cloud1/example-cloud/solr/collection1
Data: /Applications/solrLatest-cloud1/example-cloud/solr/collection1/data
Index: /Applications/solrLatest-cloud1/example-cloud/solr/collection1/data/index
Impl: org.apache.solr.core.NRTCachingDirectoryFactory

Healthcheck

Ping request handler is not configured with a healthcheck file.

After selecting the core, the central part of the screen shows Statistics and other information about the core you chose. You can define a file called `admin-extra.html` that includes links or other information you would like to display in the "Admin Extra" part of this main screen.

On the left side, under the core name, are links to other screens that display information or provide options for the specific core chosen. The core-specific options are listed below, with a link to the section of this Guide to find out more:

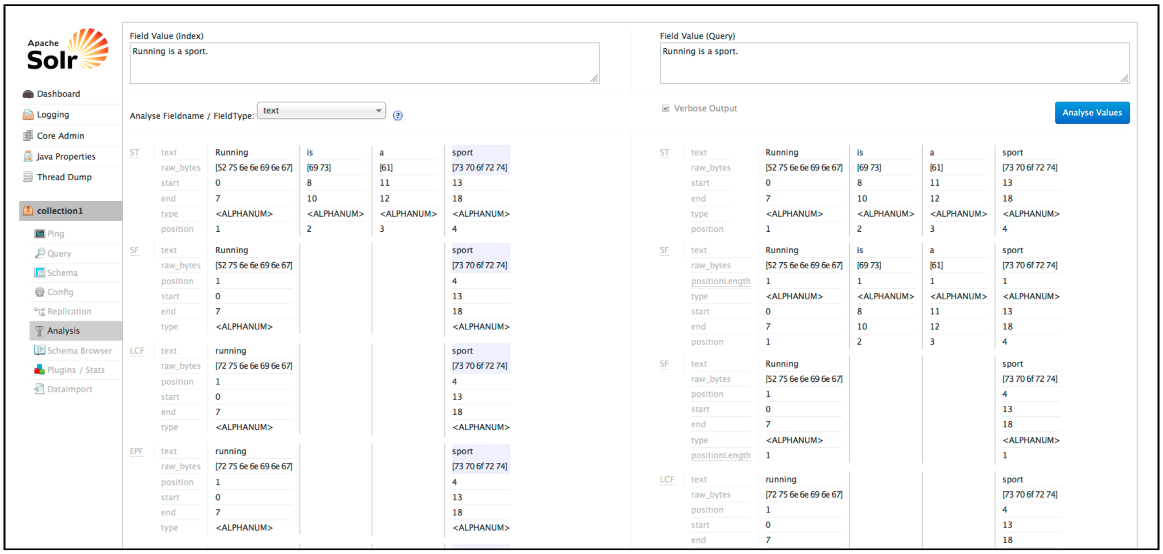
- Analysis - lets you analyze the data found in specific fields.
- Dataimport - shows you information about the current status of the Data Import Handler.
- Documents - provides a simple form allowing you to execute various Solr indexing commands directly from the browser.
- Files - shows the current core configuration files such as `solrconfig.xml` and `schema.xml`.
- Ping - lets you ping a named core and determine whether the core is active.
- Plugins/Stats - shows statistics for plugins and other installed components.
- Query - lets you submit a structured query about various elements of a core.
- Replication - shows you the current replication status for the core, and lets you enable/disable replication.
- Schema Browser - displays schema data in a browser window.

## Analysis Screen

The Analysis screen lets you inspect how data will be handled according to the field, field type and dynamic rule configurations found in `schema.xml`. You can analyze how content would be handled during indexing or during query processing and view the results separately or at the same time. Ideally, you would want content to be handled consistently, and this screen allows you to validate the settings in the field type or field analysis chains.

Enter content in one or both boxes at the top of the screen, and then choose the field or field type definitions to use for analysis.

The standard output (shown if "Verbose Output" is not checked) will display the step of the analysis and the output based on the current settings. If you click the **Verbose Output** check box, you see more information, including transformations to the input (such as, convert to lower case, strip extra characters, etc.) and the bytes, type and detailed position information. The information displayed will vary depending on the settings of the field or field type. Each step of the process is displayed in a separate section, with an abbreviation for the tokenizer or filter that is applied in that step. Hover or click on the abbreviation, and you'll see the name and path of the tokenizer or filter.



Field Value (Index)
Running is a sport.

Analyse Fieldname / FieldType: text

Field Value (Query)
Running is a sport.

☑ Verbose Output

| | | | | | |
|---|---|---|---|---|---|
| ST | text | Running | is | a | sport |
| | raw_bytes | [52 75 6e 6e 69 6e 67] | [69 73] | [61] | [73 70 6f 72 74] |
| | start | 0 | 8 | 11 | 13 |
| | end | 7 | 10 | 12 | 18 |
| | type | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> |
| | position | 1 | 2 | 3 | 4 |
| SF | text | Running | | | sport |
| | raw_bytes | [52 75 6e 6e 69 6e 67] | | | [73 70 6f 72 74] |
| | position | 1 | | | 4 |
| | start | 0 | | | 13 |
| | end | 7 | | | 18 |
| | type | <ALPHANUM> | | | <ALPHANUM> |
| LCF | text | running | | | sport |
| | raw_bytes | [72 75 6e 6e 69 6e 67] | | | [73 70 6f 72 74] |
| | position | 1 | | | 4 |
| | start | 0 | | | 13 |
| | end | 7 | | | 18 |
| | type | <ALPHANUM> | | | <ALPHANUM> |
| EPF | text | running | | | sport |
| | raw_bytes | [72 75 6e 6e 69 6e 67] | | | [73 70 6f 72 74] |
| | position | 1 | | | 4 |
| | start | 0 | | | 13 |
| | end | 7 | | | 18 |
| | type | <ALPHANUM> | | | <ALPHANUM> |

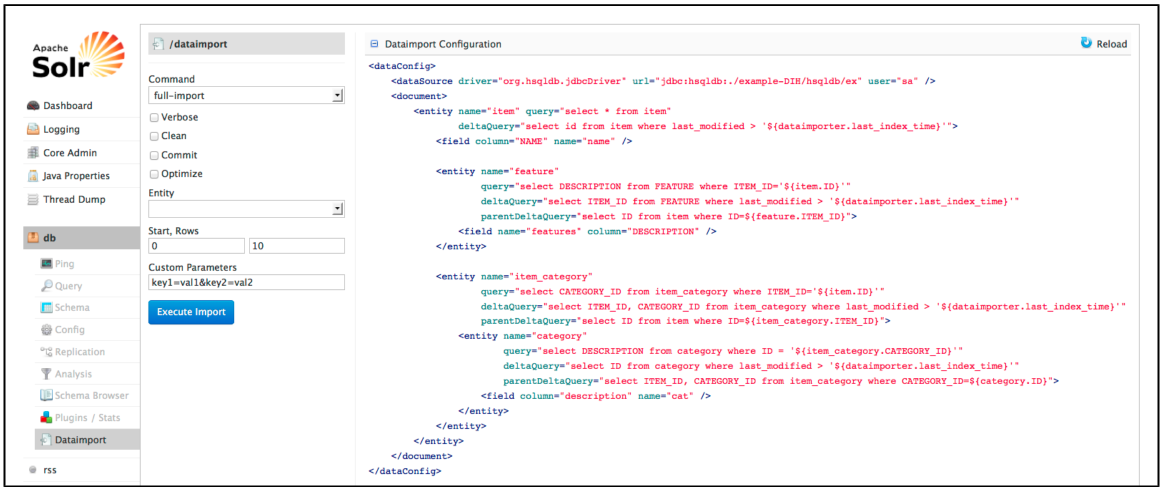
| | | | | | |
|---|---|---|---|---|---|
| ST | text | Running | is | a | sport |
| | raw_bytes | [52 75 6e 6e 69 6e 67] | [69 73] | [61] | [73 70 6f 72 74] |
| | start | 0 | 8 | 11 | 13 |
| | end | 7 | 10 | 12 | 18 |
| | type | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> |
| | position | 1 | 2 | 3 | 4 |
| SF | text | Running | is | a | sport |
| | raw_bytes | [52 75 6e 6e 69 6e 67] | [69 73] | [61] | [73 70 6f 72 74] |
| | positionLength | 1 | 1 | 1 | 1 |
| | type | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> |
| | start | 0 | 8 | 11 | 13 |
| | end | 7 | 10 | 12 | 18 |
| | position | 1 | 2 | 3 | 4 |
| SF | text | Running | | | sport |
| | raw_bytes | [52 75 6e 6e 69 6e 67] | | | [73 70 6f 72 74] |
| | position | 1 | | | 4 |
| | start | 0 | | | 13 |
| | end | 7 | | | 18 |
| | type | <ALPHANUM> | | | <ALPHANUM> |
| | positionLength | 1 | | | 1 |
| LCF | text | running | | | sport |
| | raw_bytes | [72 75 6e 6e 69 6e 67] | | | [73 70 6f 72 74] |
| | position | 1 | | | 4 |
| | start | 0 | | | 13 |
| | end | 7 | | | 18 |

In example screenshot above, several transformations are applied to the text string "Running is a sport." We've used the field "text", which has rules that remove the "is" and "a" and the word "running" has been changed to its basic form, "run". This is because we have defined the field type, `text_en` in this scenario, to remove stop words (small words that usually do not provide a great deal of context) and "stem" terms when possible to find more possible matches (this is particularly helpful with plural forms of words). If you click the question mark next to the **Analyze Fieldname/Field Type** pull-down menu, the Schema Browser window will open, showing you the settings for the field specified.

The section Understanding Analyzers, Tokenizers, and Filters describes in detail what each option is and how it may transform your data and the section Running Your Analyzer has specific examples for using the Analysis screen.

## Dataimport Screen

The Dataimport screen shows the configuration of the DataImportHandler (DIH) and allows you to start indexing data, as defined by the options selected on the screen and defined in the configuration file.



Command: full-import

□ Verbose
□ Clean
□ Commit
□ Optimize

Entity:

Start, Rows: 0 10

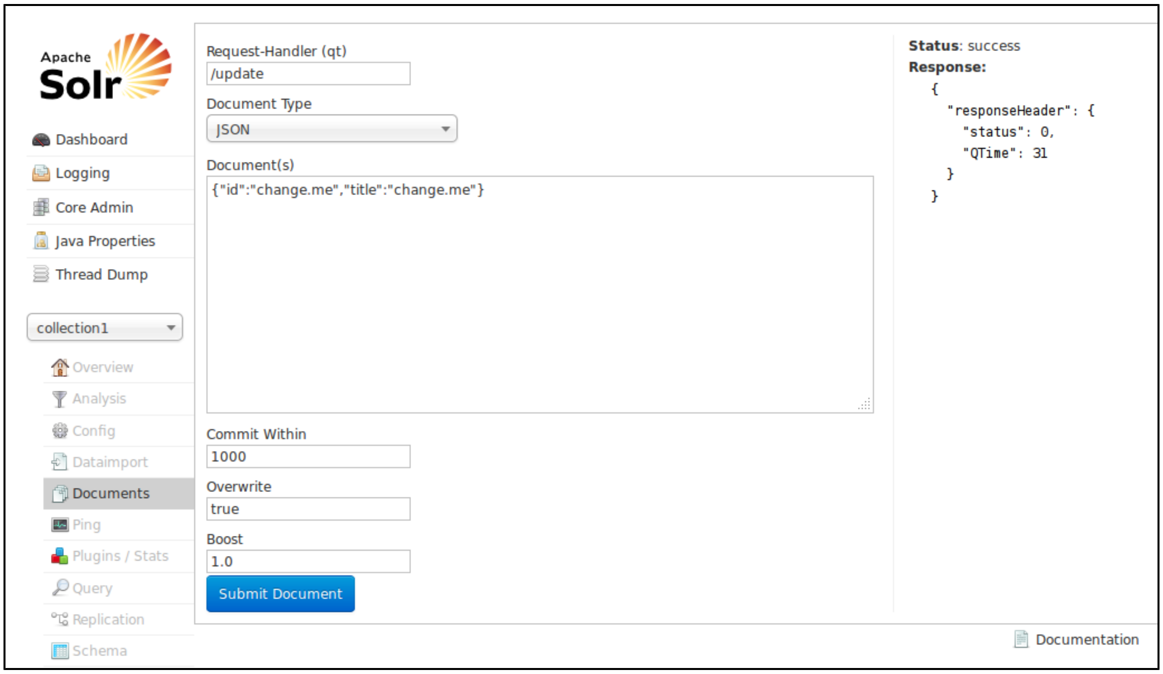
Custom Parameters: key1=val1&key2=val2

```
<dataConfig>
    <dataSource driver="org.hsqldb.jdbcDriver" url="jdbc:hsqldb:./example-DIH/hsqldb/ex" user="sa" />
    <document>
        <entity name="item" query="select * from item"
                deltaQuery="select id from item where last_modified > '${dataimporter.last_index_time}'">
            <field column="NAME" name="name" />

            <entity name="feature"
                    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"
                    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=${item_category.ITEM_ID}">
                <entity name="category"
                        query="select DESCRIPTION from category where ID = '${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>
</dataConfig>
```

The configuration file defines the location of the data and how to perform the SQL queries for the data you want. The options on the screen control how the data is imported to Solr. For more information about data importing with DIH, see the section on Uploading Structured Data Store Data with the Data Import Handler.

## Documents Screen

The Documents screen provides a simple form allowing you to execute various Solr indexing commands in a variety of formats directly from the browser.



Request-Handler (qt): /update

Document Type: JSON

Document(s):

```
{"id":"change.me","title":"change.me"}
```

Commit Within: 1000

Overwrite: true

Boost: 1.0

**Status**: success

**Response:**

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 31
  }
}
```

The screen allows you to:

- Copy documents in JSON, CSV or XML and submit them to the index
- Upload documents (in JSON, CSV or XML)
- Construct documents by selecting fields and field values

The first step is to define the RequestHandler to use (aka, 'qt'). By default `/update` will be defined. To use Solr Cell, for example, change the request handler to `/update/extract`.

Then choose the Document Type to define the type of document to load. The remaining parameters will change depending on the document type selected.

### *JSON*

When using the JSON document type, the functionality is similar to using a requestHandler on the command line. Instead of putting the documents in a curl command, they can instead be input into the Document entry box. The document structure should still be in proper JSON format.

Then you can choose when documents should be added to the index (Commit Within), whether existing documents should be overwritten with incoming documents with the same id (if this is not **true**, then the incoming documents will be dropped), and, finally, if a document boost should be applied.

This option will only add or overwrite documents to the index; for other update tasks, see the #Solr Command option .

### *CSV*

When using the CSV document type, the functionality is similar to using a requestHandler on the command line. Instead of putting the documents in a curl command, they can instead be input into the Document entry box. The document structure should still be in proper CSV format, with columns delimited and one row per document.

Then you can choose when documents should be added to the index (Commit Within), and whether existing documents should be overwritten with incoming documents with the same id (if this is not **true**, then the incoming documents will be dropped).

#### *Document Builder*

The Document Builder provides a wizard-like interface to enter fields of a document

#### *File Upload*

The File Upload option allows choosing a prepared file and uploading it. If using only `/update` for the Request-Handler option, you will be limited to XML, CSV, and JSON.

However, to use the ExtractingRequestHandler (aka Solr Cell), you can modify the Request-Handler to `/update/extract`. You must have this defined in your `solrconfig.xml` file, with your desired defaults. You should also update the `&literal.id` shown in the Extracting Req. Handler Params so the file chosen is given a unique id.

Then you can choose when documents should be added to the index (Commit Within), and whether existing documents should be overwritten with incoming documents with the same id (if this is not **true**, then the incoming documents will be dropped).

#### *Solr Command*

The Solr Command option allows you use XML or JSON to perform specific actions on documents, such as defining documents to be added or deleted, updating only certain fields of documents, or commit and optimize commands on the index.

The documents should be structured as they would be if using `/update` on the command line.

#### *XML*

When using the XML document type, the functionality is similar to using a requestHandler on the command line. Instead of putting the documents in a curl command, they can instead be input into the Document entry box. The document structure should still be in proper Solr XML format, with each document separated by `<doc>` tags and each field defined.

Then you can choose when documents should be added to the index (Commit Within), and whether existing documents should be overwritten with incoming documents with the same id (if this is not **true**, then the incoming documents will be dropped).

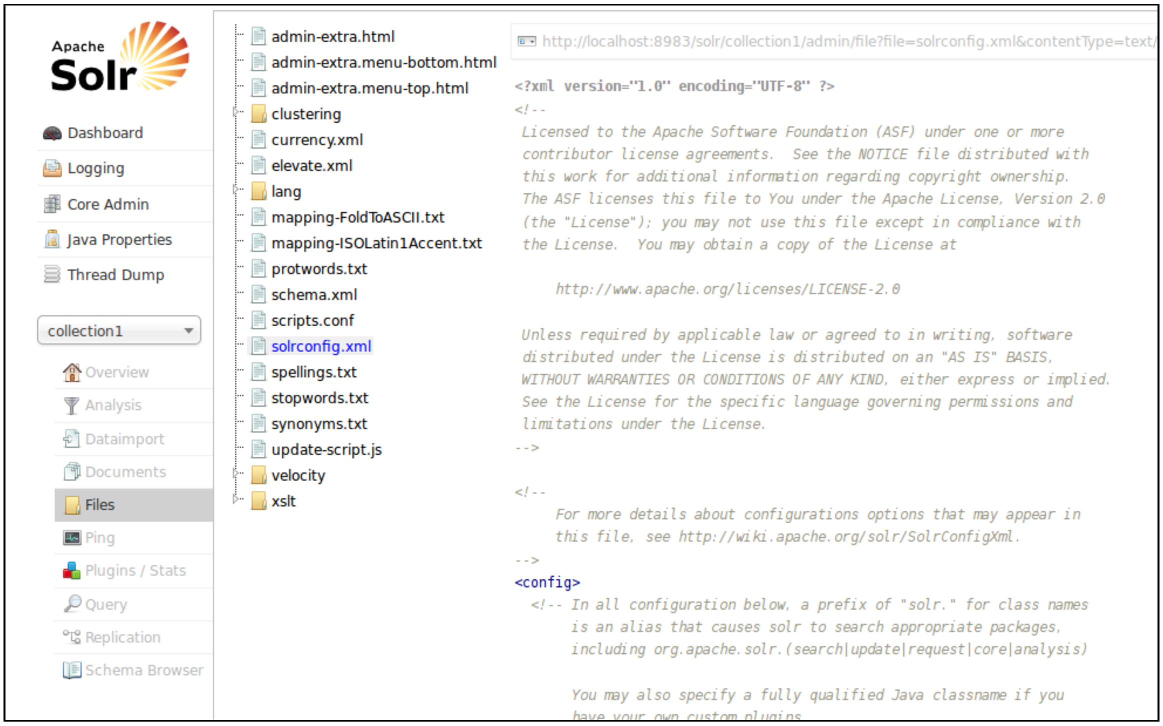
This option will only add or overwrite documents to the index; for other update tasks, see the #Solr Command option .

#### Related Topics

- Uploading Data with Index Handlers
- Uploading Data with Solr Cell using Apache Tika

## Files Screen

The Files screen lets you browse & view the various configuration files (such `solrconfig.xml` and `schema.xml`) for the core you selected.



http://localhost:8983/solr/collection1/admin/file?file=solrconfig.xml&contentType=text/

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
 Licensed to the Apache Software Foundation (ASF) under one or more
 contributor license agreements.  See the NOTICE file distributed with
 this work for additional information regarding copyright ownership.
 The ASF licenses this file to You under the Apache License, Version 2.0
 (the "License"); you may not use this file except in compliance with
 the License.  You may obtain a copy of the License at

     http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
-->

<!--
     For more details about configurations options that may appear in
     this file, see http://wiki.apache.org/solr/SolrConfigXml.
-->
<config>
  <!-- In all configuration below, a prefix of "solr." for class names
       is an alias that causes solr to search appropriate packages,
       including org.apache.solr.(search|update|request|core|analysis)

       You may also specify a fully qualified Java classname if you
       have your own custom plugins.
```

While the `solrconfig.xml` defines the behaviour of Solr as it indexes content and responds to queries, the `schema.xml` allows you to define the types of data in your content (field types), the fields your documents will be broken into, and any dynamic fields that should be generated based on patterns of field names in the incoming documents. Any other configuration files are used depending on how they are referenced in either `solrconfig.xml` or `schema.xml`.

Configuration files cannot be edited with this screen, so a text editor of some kind must be used.

This screen is related to the Schema Browser Screen, in that they both can display information from the schema, but the Schema Browser provides a way to drill into the analysis chain and displays linkages between field types, fields, and dynamic field rules.

Many of the options defined in `solrconfig.xml` and `schema.xml` are described throughout the rest of this Guide. In particular, you will want to review these sections:

- Indexing and Basic Data Operations
- Searching
- The Well-Configured Solr Instance
- Documents, Fields, and Schema Design

## Ping

Choosing Ping under a core name issues a `ping` request to check whether a server is up.

Ping is configured using a `requestHandler` in the `solrconfig.xml` file:

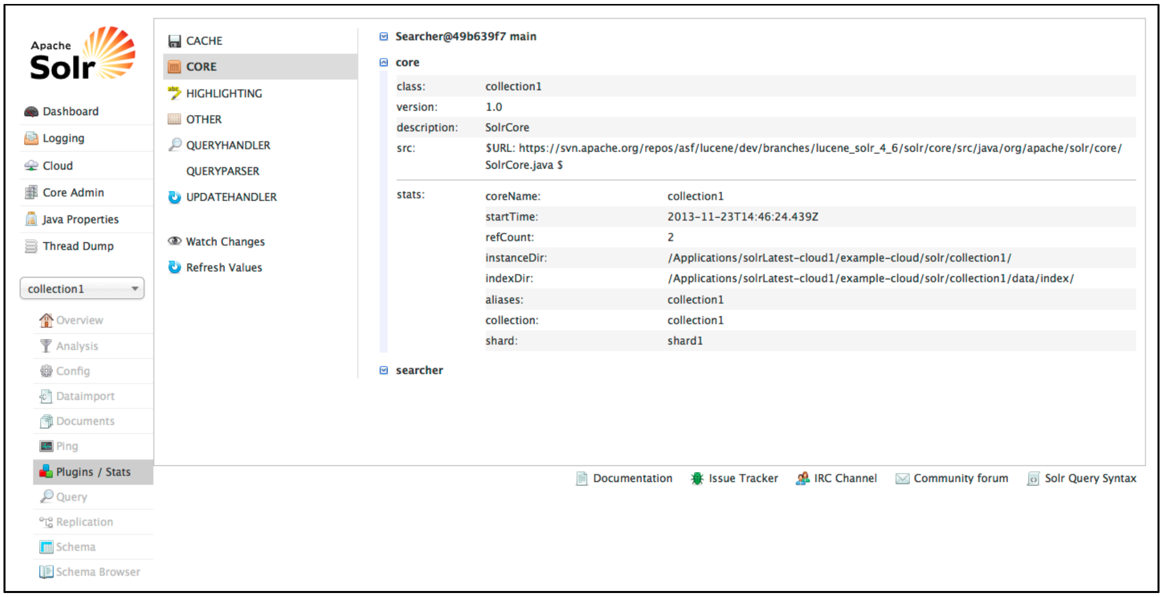
```
<!-- ping/healthcheck -->
  <requestHandler name="/admin/ping" class="solr.PingRequestHandler">
    <lst name="invariants">
      <str name="q">solrpingquery</str>
    </lst>
    <lst name="defaults">
      <str name="echoParams">all</str>
    </lst>
    <!-- An optional feature of the PingRequestHandler is to configure the
         handler with a "healthcheckFile" which can be used to enable/disable
         the PingRequestHandler.
         relative paths are resolved against the data dir
      -->
    <!-- <str name="healthcheckFile">server-enabled.txt</str> -->
  </requestHandler>
```

The Ping option doesn't open a page, but the status of the request can be seen on the core overview page shown when clicking on a collection name. The length of time the request has taken is displayed next to the Ping option, in milliseconds.

## Plugins & Stats Screen

The Plugins screen shows information and statistics about Solr's status and performance. You can find information about the performance of Solr's caches, the state of Solr's searchers, and the configuration of searchHandlers and requestHandlers.

Choose an area of interest on the right, and then drill down into more specifics by clicking on one of the names that appear in the central part of the window. In this example, we've chosen to look at the Searcher stats, from the Core area:



Searcher@49b639f7 main

core

| | | |
|---|---|---|
| class: | collection1 | |
| version: | 1.0 | |
| description: | SolrCore | |
| src: | $URL: https://svn.apache.org/repos/asf/lucene/dev/branches/lucene_solr_4_6/solr/core/src/java/org/apache/solr/core/ SolrCore.java $ | |
| stats: | coreName: | collection1 |
| | startTime: | 2013-11-23T14:46:24.439Z |
| | refCount: | 2 |
| | instanceDir: | /Applications/solrLatest-cloud1/example-cloud/solr/collection1/ |
| | indexDir: | /Applications/solrLatest-cloud1/example-cloud/solr/collection1/data/index/ |
| | aliases: | collection1 |
| | collection: | collection1 |
| | shard: | shard1 |

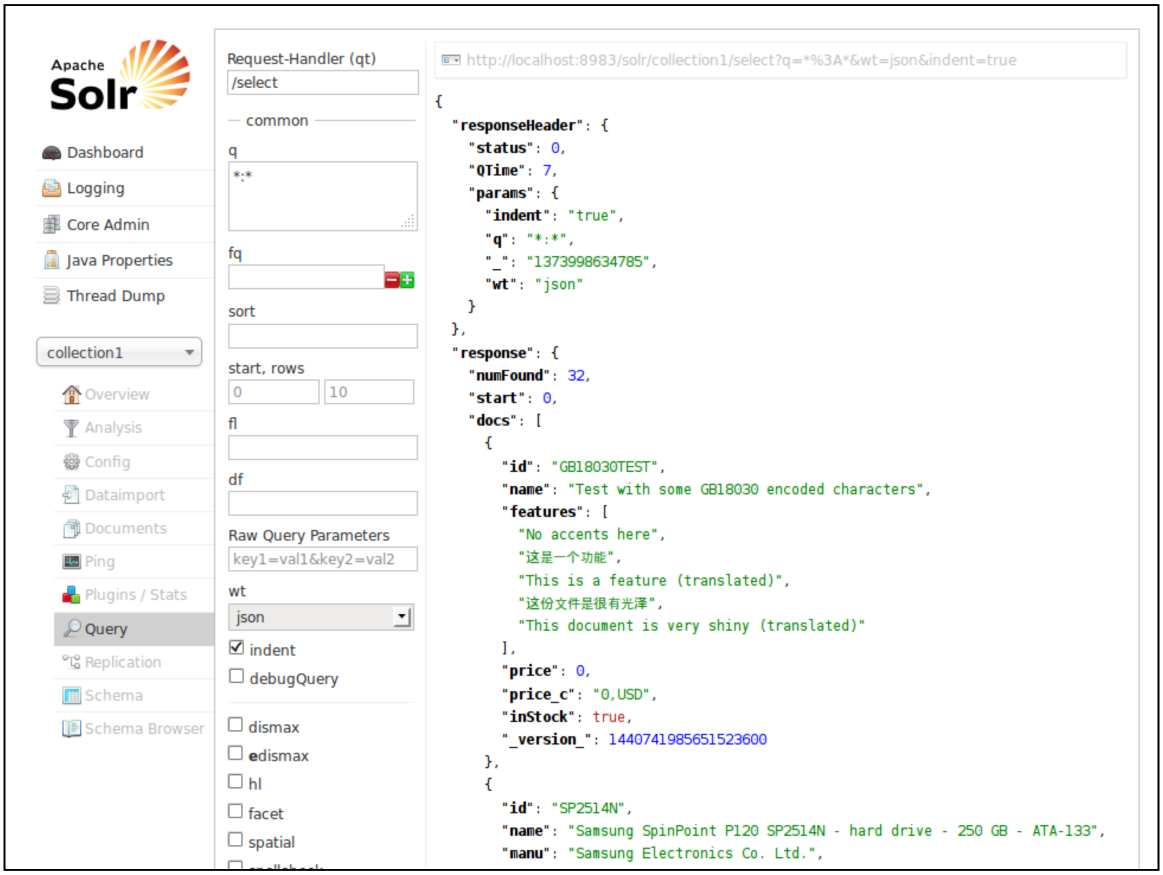
searcher

*Searcher Statistics*

The display is a snapshot taken when the page is loaded. You can get updated status by choosing to either **Watch Changes** or **Refresh Values**. Watching the changes will highlight those areas that have changed, while refreshing the values will reload the page with updated information.

## Query Screen

You can use **Query**, shown under the name of each core, to submit a search query to a Solr server and analyze the results. In the example in the screenshot, a query has been submitted, and the screen shows the query results sent to the browser as JSON.



Request-Handler (qt): /select

common

q: *:*

fq:

sort:

start, rows: 0 | 10

fl:

df:

Raw Query Parameters: key1=val1&key2=val2

wt: json

☑ indent
☐ debugQuery

☐ dismax
☐ edismax
☐ hl
☐ facet
☐ spatial

http://localhost:8983/solr/collection1/select?q=*%3A*&wt=json&indent=true

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 7,
    "params": {
      "indent": "true",
      "q": "*:*",
      "_": "1373998634785",
      "wt": "json"
    }
  },
  "response": {
    "numFound": 32,
    "start": 0,
    "docs": [
      {
        "id": "GB18030TEST",
        "name": "Test with some GB18030 encoded characters",
        "features": [
          "No accents here",
          "这是一个功能",
          "This is a feature (translated)",
          "这份文件是很有光泽",
          "This document is very shiny (translated)"
        ],
        "price": 0,
        "price_c": "0,USD",
        "inStock": true,
        "_version_": 1440741985651523600
      },
      {
        "id": "SP2514N",
        "name": "Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133",
        "manu": "Samsung Electronics Co. Ltd.",
```

The query was sent to a core named "collection1". We used Solr's default query for this screen (as defined in `solrconfig.xml`), which is `*:*`. This query will find all records in the index for this core. We kept the other defaults, but the table below explains these options, which are also covered in detail in later parts of this Guide.

The response is shown to the right of the form. Requests to Solr are simply HTTP requests, and the query submitted is shown in light type above the results; if you click on this it will open a new browser window with just this request and response (without the rest of the Solr Admin UI). The rest of the response is shown in JSON, which is part of the request (see the `wt=json` part at the end).

The response has at least two sections, but may have several more depending on the options chosen. The two sections it always has are the `responseHeader` and the `response`. The `responseHeader` includes the status of the search (`status`), the processing time (`QTime`), and the parameters (`params`) that were used to process the query.

The `response` includes the documents that matched the query, in `doc` sub-sections. The fields return depend on the parameters of the query (and the defaults of the request handler used). The number of results is also included in this section.

This screen allows you to experiment with different query options, and inspect how your documents were indexed. The query parameters available on the form are some basic options that most users want to have available, but there are dozens more available which could be simply added to the basic request by hand (if opened in a browser). The table below explains the parameters available:

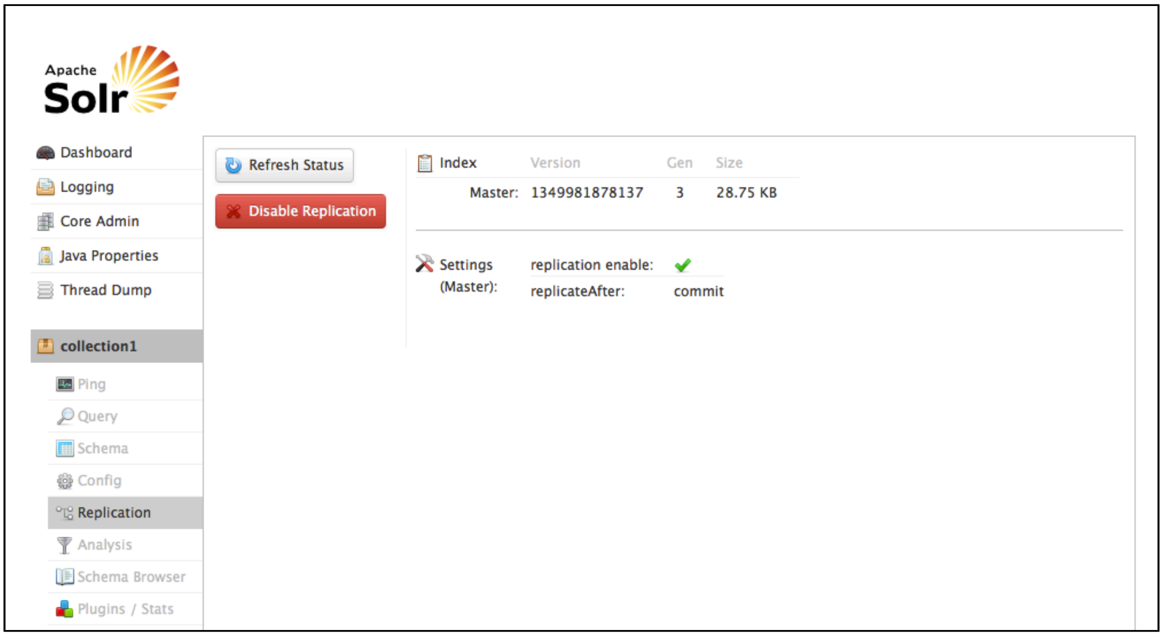
| Field | Description |
|---|---|
| Request-handler (qt) | Specifies the query handler for the request. If a query handler is not specified, Solr processes the response with the standard query handler. |
| q | The query event. See Searching for an explanation of this parameter. |
| fq | The filter queries. See Common Query Parameters for more information on this parameter. |
| sort | Sorts the response to a query in either ascending or descending order based on the response's score or another specified characteristic. |
| start, rows | `start` is 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 Searching. `rows` is the number of rows to return. |
| fl | Defines the fields to return for each document. You can explicitly list the stored fields you want to have returned by separating them with either a comma or a space. In Solr 4, the results of functions can also be included in the `fl` list. |
| wt | Specifies the Response Writer to be used to format the query response. Defaults to XML if not specified. |
| indent | Click this button to request that the Response Writer use indentation to make the responses more readable. |
| debugQuery | Click this button to augment 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. |
| dismax | Click this button to enable the Dismax query parser. See The DisMax Query Parser for further information. |
| edismax | Click this button to enable the Extended query parser. See The Extended DisMax Query Parser for further information. |
| hl | Click this button to enable highlighting in the query response. See Highlighting for more information. |
| facet | Enables faceting, the arrangement of search results into categories based on indexed terms. See Faceting for more information. |
| spatial | Click to enable using location data for use in spatial or geospatial searches. See Spatial Search for more information. |
| spellcheck | Click this button to enable the Spellchecker, which provides inline query suggestions based on other, similar, terms. See Spell Checking for more information. |

**Related Topics**

- Searching

## Replication Screen

The Replication screen shows you the current replication state for the named core you have specified. In Solr, replication is for the index only. SolrCloud has supplanted much of this functionality, but if you are still using index replication, you can use this screen to see the replication state:



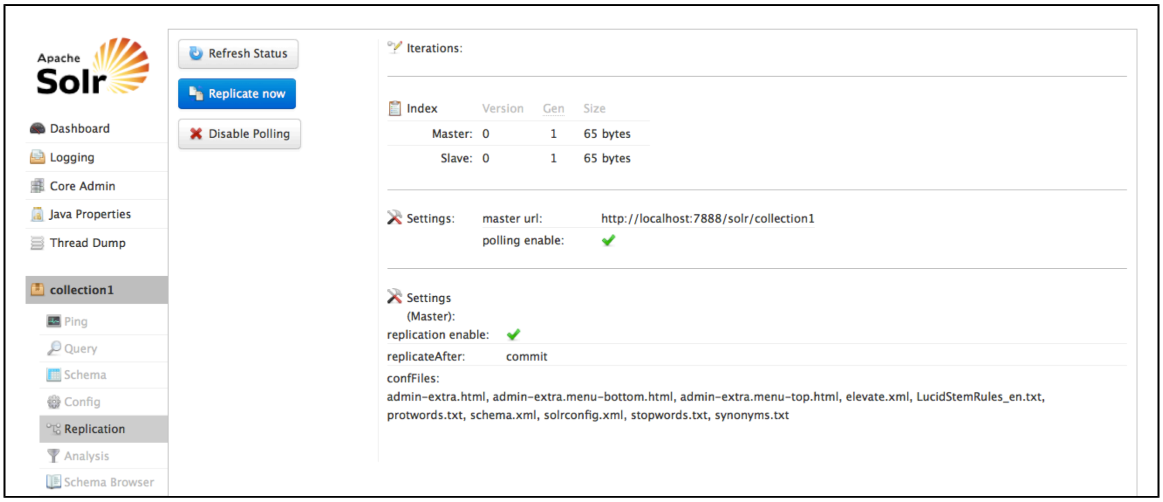
| Index | Version | Gen | Size |
|---|---|---|---|
| Master: | 1349981878137 | 3 | 28.75 KB |

Settings (Master):

| | |
|---|---|
| replication enable: | ✓ |
| replicateAfter: | commit |

In this example, replication is enabled and will be done after each commit. Because this server is the Master, it is showing only the config settings for the master. On the master, you can disable replication by clicking the **Disable Replication** button.

In Solr, the replication is initiated by the slave servers so there is more value by looking at the Replication screen on the slave nodes. This screenshot shows the Replication screen for a slave:



Iterations:

| Index | Version | Gen | Size |
|---|---|---|---|
| Master: | 0 | 1 | 65 bytes |
| Slave: | 0 | 1 | 65 bytes |

Settings:

| | |
|---|---|
| master url: | http://localhost:7888/solr/collection1 |
| polling enable: | ✓ |

Settings (Master):

replication enable: ✓

replicateAfter: commit

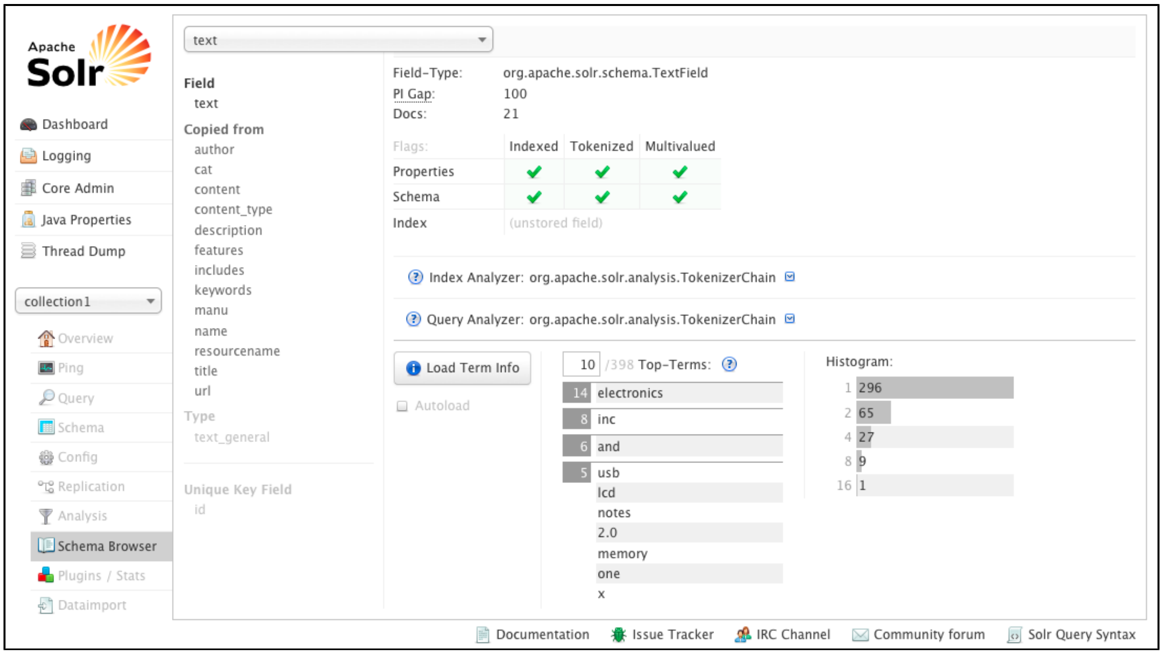
confFiles:
admin-extra.html, admin-extra.menu-bottom.html, admin-extra.menu-top.html, elevate.xml, LucidStemRules_en.txt, protwords.txt, schema.xml, solrconfig.xml, stopwords.txt, synonyms.txt

You can click the **Refresh Status** button to show the most current replication status, or choose to get a new snapshot from the master server.

More details on how to configure replication is available in the section called Index Replication.

## Schema Browser Screen

The Schema Browser screen lets you see schema data in a browser window. If you have accessed this window from the Analysis screen, it will be opened to a specific field, dynamic field rule or field type. If there is nothing chosen, use the pull-down menu to choose the field or field type.



**Field**
text

**Copied from**
author
cat
content
content_type
description
features
includes
keywords
manu
name
resourcename
title
url

Type
text_general

Unique Key Field
id

Field-Type: org.apache.solr.schema.TextField
PI Gap: 100
Docs: 21

| Flags: | Indexed | Tokenized | Multivalued |
|---|---|---|---|
| Properties | ✓ | ✓ | ✓ |
| Schema | ✓ | ✓ | ✓ |
| Index | (unstored field) | | |

Index Analyzer: org.apache.solr.analysis.TokenizerChain

Query Analyzer: org.apache.solr.analysis.TokenizerChain

10 /398 Top-Terms:

| | |
|---|---|
| 14 | electronics |
| 8 | inc |
| 6 | and |
| 5 | usb |
| | lcd |
| | notes |
| | 2.0 |
| | memory |
| | one |
| | x |

Histogram:

| | |
|---|---|
| 1 | 296 |
| 2 | 65 |
| 4 | 27 |
| 8 | 9 |
| 16 | 1 |

The screen provides a great deal of useful information about each particular field. In the example above, we have chosen the `text` field. On the right side of the center window, we see the field name, and a list of fields that populate this field because they are defined to be copied to the `text` field. Click on one of those field names, and you can see the definitions for that field. We can also see the field type, which would allow us to inspect the type definitions as well.

In the left part of the center window, we see the field type again, and the defined properties for the field. We can also see how many documents have populated this field. Then we see the analyzer used for indexing and query processing. Click the icon to the left of either of those, and you'll see the definitions for the tokenizers and/or filters that are used. The output of these processes is the information you see when testing how content is handled for a particular field with the Analysis Screen.

Under the analyzer information is a button to **Load Term Info**. Clicking that button will show the top *N* terms that are in the index for that field. Click on a term, and you will be taken to the Query Screen to see the results of a query of that term in that field. If you want to always see the term information for a field, choose **Autoload** and it will always appear when there are terms for a field. A histogram shows the number of terms with a given frequency in the field.

# Documents, Fields, and Schema Design

This section discusses how Solr organizes its data into documents and fields, as well as how to work with the Solr schema file, `schema.xml`. It includes the following topics:

Overview of Documents, Fields, and Schema Design: An introduction to the concepts covered in this section.

Solr Field Types: Detailed information about field types in Solr, including the field types in the default Solr schema.

Defining Fields: Describes how to define fields in Solr.

Copying Fields: Describes how to populate fields with data copied from another field.

Dynamic Fields: Information about using dynamic fields in order to catch and index fields that do not exactly conform to other field definitions in your schema.

Schema API: Use curl commands to read various parts of a schema or create new fields and copyField rules.

Other Schema Elements: Describes other important elements in the Solr schema: Unique Key, Default Search Field, and the Query Parser Operator.

Putting the Pieces Together: A higher-level view of the Solr schema and how its elements work together.

DocValues: Describes how to create a docValues index for faster lookups.

Schemaless Mode: Automatically add previously unknown schema fields using value-based field type guessing.

# Overview of Documents, Fields, and Schema Design

The fundamental premise of Solr is simple. You give it a lot 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.

### How Solr Sees the World

Solr's basic unit of information is a *document*, which is a set of data 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 rather than a floating point number, for example), but if you define your fields correctly, Solr will be able to interpret them correctly and 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.

### 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. Understanding Analyzers, Tokenizers, and Filters is a detailed description of field analysis.

## Solr Field Types

The field type defines how Solr should interpret data in a field and how the field can be queried. There are many field types included with Solr by default, and they can also be defined locally.

Topics covered in this section:

- Field Type Definitions and Properties
- Field Types Included with Solr
- Working with Currencies and Exchange Rates
- Working with Dates
- Working with Enum Fields
- Working with External Files and Processes
- Field Properties by Use Case

### Related Topics

- SchemaXML-DataTypes
- FieldType Javadoc

## Field Type Definitions and Properties

A field type definition can include four types of information:

- The name of the field type (mandatory)
- An implementation class name (mandatory)
  - If the field type is `TextField`, a description of the field analysis for the field type
- Field type properties - depending on the implementation class, some properties may be mandatory.

### Field Type Definitions in `schema.xml`

Field types are defined in `schema.xml`, with the `types` element. Each field type is defined between `fieldType` elements. Here is an example of a field type definition for a type called `text_general`:

```
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
enablePositionIncrements="true" />
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymFilterFactory" synonyms="index_synonyms.txt"
ignoreCase="true" expand="false"/>
    -->
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
enablePositionIncrements="true" />
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

The first line in the example above contains the field type name, `text_general`, and the name of the implementing class, `solr.TextField`. The rest of the definition is about field analysis, described in Understanding Analyzers, Tokenizers, and Filters.

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.`.

### Field Type Properties

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

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

The properties that can be specified for a given field type fall into three major categories:

- Properties specific to the field type's class.

- General Properties Solr supports for any field type.
- Field Default Properties that can be specified on the field type that will be inherited by fields that use this type instead of the default behavior.

***General Properties***

| Property | Description | Values |
|---|---|---|
| name | The name of the fieldType.  This value gets used in field definitions, in the "type" attribute.  It is strongly recommended that names consist of alphanumeric or underscore characters only and not start with a digit.  This is not currently strictly enforced. | |
| class | The class name that gets used to store and index the data for this type.  Note that you may prefix included class names with "solr." and Solr will automatically figure out which packages to search for the class - so "solr.TextField" will work.  If you are using a third-party class, you will probably need to have a fully qualified class name.  The fully qualified equivalent for "solr.TextField" is "org.apache.solr.schema.TextField". | |
| positionIncrementGap | For multivalued fields, specifies a distance between multiple values, which prevents spurious phrase matches | integer |
| autoGeneratePhraseQueries | For text fields. If true, Solr automatically generates phrase queries for adjacent terms. If false, terms must be enclosed in double-quotes to be treated as phrases. | true or false |
| docValuesFormat | Defines a custom `DocValuesFormat` to use for fields of this type. This requires that a schema-aware codec, such as the `SchemaCodecFactory` has been configured in solrconfig.xml. | n/a |
| postingsFormat | Defines a custom `PostingsFormat` to use for fields of this type. This requires that a schema-aware codec, such as the `SchemaCodecFactory` has been configured in solrconfig.xml. | n/a |

> ⓘ Lucene index back-compatibility is only supported for the default codec. If you choose to customize the `postingsFormat` or `docValuesFormat` in your schema.xml, upgrading to a future version of Solr may require you to either switch back to the default codec and optimize your index to rewrite it into the default codec before upgrading, or re-build your entire index from scratch after upgrading.

***Field Default Properties***

| 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 |

| | | |
|---|---|---|
| docValues | If true, the value of the field will be put in a column-oriented DocValues structure | true or false |
| sortMissingFirst<br>sortMissingLast | Control the placement of documents when a sort field is not present. As of Solr 3.5, these work for all numeric fields, including Trie and date fields. | true or false |
| multiValued | If true, indicates that a single document might contain multiple values for this field type | true or false |
| 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). Defaults to true for all primitive (non-analyzed) field types, such as int, float, data, bool, and string. 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 |
| omitPositions | Similar to `omitTermFreqAndPositions` but preserves term frequency information | true or false |
| termVectors<br>termPositions<br>termOffsets | These options instruct Solr to maintain full term vectors for each document, optionally including the position and offset information for each term occurrence in those vectors. These can be used to accelerate highlighting and other ancillary functionality, but impose a substantial cost in terms of index size. They are not necessary for typical uses of Solr | true or false |
| required | Instructs Solr to reject any attempts to add a document which does not have a value for this field. This property defaults to false. | true or false |

## Field Types Included with Solr

The following table lists the field types that are available in Solr. 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. *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| BCDLongField | Binary-coded decimal long integer. *This field has been deprecated and will be removed in Solr 5.0, use TrieLongField instead.* |
| BCDStrField | Binary-coded decimal string. *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| 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 a byte (an 8-bit signed integer). *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| CollationField | Supports Unicode collation for sorting and range queries. ICUCollationField is a better choice if you can use ICU4J. See the section Unicode Collation. |
| CurrencyField | Supports currencies and exchange rates. See the section Working with Currencies and Exchange Rates. |
| DateField | Represents a point in time with millisecond precision. See the section Working with Dates. *This field has been deprecated and will be removed in Solr 5.0, use TrieDateField instead.* |
| DoubleField | Double (64-bit IEEE floating point). *This field has been deprecated and will be removed in Solr 5.0, use TrieDoubleField instead.* |
| ExternalFileField | Pulls values from a file on disk. See the section Working with External Files and Processes. |
| EnumField | Allows defining an enumerated set of values which may not be easily sorted by either alphabetic or numeric order (such as a list of severities, for example). This field type takes a configuration file, which lists the proper order of the field values. See the section Working with Enum Fields for more information. |
| FloatField | Floating point (32-bit IEEE floating point). *This field has been deprecated and will be removed in Solr 5.0, use TrieFloatField instead.* |
| ICUCollationField | Supports Unicode collation for sorting and range queries. See the section Unicode Collation. |
| IntField | Integer (32-bit signed integer). *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| LatLonType | Spatial Search: a latitude/longitude coordinate pair. The latitude is specified first in the pair. |
| LongField | Long integer (64-bit signed integer). *This field has been deprecated and will be removed in Solr 5.0, use TrieLongField instead.* |
| PointType | Spatial Search: An arbitrary n-dimensional point, useful for searching sources such as blueprints or CAD drawings. |

| | |
|---|---|
| PreAnalyzedField | Provides a way to send to Solr serialized token streams, optionally with independent stored values of a field, and have this information stored and indexed without any additional text processing. Useful if you want to submit field content that was already processed by some existing external text processing pipeline (e.g. tokenized, annotated, stemmed, inserted synonyms, etc.), while using all the rich attributes that Lucene's `TokenStream` provides via token attributes. |
| 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. *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| SortableDoubleField | The Sortable fields provide correct numeric sorting. *This field has been deprecated and will be removed in Solr 5.0, use TrieDoubleField instead.* |
| SortableFloatField | Numerically sorted floating point. *This field has been deprecated and will be removed in Solr 5.0, use TrieFloatField instead.* |
| SortableIntField | Numerically sorted integer. *This field has been deprecated and will be removed in Solr 5.0, use TrieIntField instead.* |
| SortableLongField | Numerically sorted long integer. *This field has been deprecated and will be removed in Solr 5.0, use TrieLongField instead.* |
| SpatialRecursivePrefixTreeFieldType | (RPT for short) Spatial Search: Accepts latitude comma longitude strings or other shapes in WKT format. |
| StrField | String (UTF-8 encoded string or Unicode). |
| TextField | Text, usually multiple words or tokens. |
| TrieDateField | Date field. Represents a point in time with millisecond precision. See the section Working with Dates. `precisionStep="0"` enables efficient date sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |
| TrieDoubleField | Double field (64-bit IEEE floating point). `precisionStep="0"` enables efficient numeric sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |
| TrieField | If this field type is used, a "type" attribute must also be specified, valid values are: `integer`, `long`, `float`, `double`, `date`. Using this field is the same as using any of the Trie fields. `precisionStep="0"` enables efficient numeric sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |
| TrieFloatField | Floating point field (32-bit IEEE floating point). `precisionStep="0"` enables efficient numeric sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |

| | |
|---|---|
| TrieIntField | Integer field (32-bit signed integer). `precisionStep="0"` enables efficient numeric sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |
| TrieLongField | Long field (64-bit signed integer). `precisionStep="0"` enables efficient numeric sorting and minimizes index size; `precisionStep="8"` (the default) enables efficient range queries. |
| UUIDField | Universally Unique Identifier (UUID). Pass in a value of "NEW" and Solr will create a new UUID. **Note**: configuring a UUIDField instance with a default value of "NEW" is not advisable for most users when using SolrCloud (and not possible if the UUID value is configured as the unique key field) since the result will be that each replica of each document will get a unique UUID value. Using UUIDUpdateProcessorFactory to generate UUID values when documents are added is recommended instead. |

The `MultiTermAwareComponent` has been added to relevant `solr.TextField` entries in `schema.xml` (e.g., wildcards, regex, prefix, range, etc.) to allow automatic lowercasing for multi-term queries.

Further, you can optionally specify a multi-term analyzer in field types in your schema: `<analyzer type="multiterm">`; if you don't do this, `analyzer` will process the fields according to their specific attributes.

## Working with Currencies and Exchange Rates

The `currency` FieldType provides support for monetary values to Solr/Lucene with query-time currency conversion and exchange rates. The following features are supported:

- Point queries
- Range queries
- Function range queries (new in Solr 4.2)
- Sorting
- Currency parsing by either currency code or symbol
- Symmetric & asymmetric exchange rates (asymmetric exchange rates are useful if there are fees associated with exchanging the currency)

### Configuring Currencies

The `currency` field type is defined in `schema.xml`. This is the default configuration of this type:

```
<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
           defaultCurrency="USD" currencyConfig="currency.xml" />
```

In this example, we have defined the name and class of the field type, and defined the `defaultCurrency` as "USD", for U.S. Dollars. We have also defined a `currencyConfig` to use a file called "currency.xml". This is a file of exchange rates between our default currency to other currencies. There is an alternate implementation that would allow regular downloading of currency data. See #Exchange Rates below for more.

At indexing time, money fields can be indexed in a native currency. For example, if a product on an e-commerce site is listed in Euros, indexing the price field as "1000,EUR" will index it appropriately. The price should be separated from the currency by a comma, and the price must be encoded with a floating point value (a decimal point).

During query processing, range and point queries are both supported.

### Exchange Rates

You configure exchange rates by specifying a provider. Natively, two provider types are supported: `FileExchangeRateProvider` or `OpenExchangeRatesOrgProvider`.

#### *FileExchangeRateProvider*

This provider requires you to provide a file of exchange rates. It is the default, meaning that to use this provider you only need to specify the file path and name as a value for `currencyConfig` in the definition for this type.

There is a sample `currency.xml` file included with Solr, found in the same directory as the `schema.xml` file. Here is a small snippet from this file:

```
<currencyConfig version="1.0">
  <rates>
    <!-- Updated from http://www.exchangerate.com/ at 2011-09-27 -->
    <rate from="USD" to="ARS" rate="4.333871" comment="ARGENTINA Peso" />
    <rate from="USD" to="AUD" rate="1.025768" comment="AUSTRALIA Dollar" />
    <rate from="USD" to="EUR" rate="0.743676" comment="European Euro" />
    <rate from="USD" to="CAD" rate="1.030815" comment="CANADA Dollar" />

    <!-- Cross-rates for some common currencies -->
    <rate from="EUR" to="GBP" rate="0.869914" />
    <rate from="EUR" to="NOK" rate="7.800095" />
    <rate from="GBP" to="NOK" rate="8.966508" />

    <!-- Asymmetrical rates -->
    <rate from="EUR" to="USD" rate="0.5" />
  </rates>
</currencyConfig>
```

#### *OpenExchangeRatesOrgProvider*

With Solr 4, you can configure Solr to download exchange rates from OpenExchangeRates.Org, with updates rates between USD and 158 currencies hourly. These rates are symmetrical only.

In this case, you need to specify the `providerClass` in the definitions for the field type. Here is an example:

```
<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
           providerClass="solr.OpenExchangeRatesOrgProvider"
           refreshInterval="60"
           ratesFileLocation="http://internal.server/rates.json"/>
```

The `refreshInterval` is minutes, so the above example will download the newest rates every 60 minutes.

## Working with Dates

### Date Formatting

Solr's `TrieDateField` (and deprecated `DateField`) represents a point in time with millisecond precision. The format used is a restricted form of the canonical representation of `dateTime` in the XML Schema specification:

`YYYY-MM-DDThh: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.
- `Z` is a literal 'Z' character indicating that this string representation of the date is in UTC

Note that no time zone can be specified; the String representations of dates is always expressed in Coordinated Universal Time (UTC). Here is an example value:

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

You can optionally include fractional seconds if you wish, although any precision beyond milliseconds will be ignored. Here are examples value with sub-seconds include:

- `1972-05-20T17:33:18.772Z`
- `1972-05-20T17:33:18.77Z`
- `1972-05-20T17:33:18.7Z`

### Date Math

Solr's date field types also supports *date math* expressions, which makes it easy to create times relative to fixed moments in time, include the current time which can be represented using the special value of "`NOW`".

#### *Date Math Syntax*

Date math expressions consist either adding some quantity of time in a specified unit, or rounding the current time by a specified unit. expressions can be chained and are evaluated left to right.

For example: this represents a point in time two months from now:

```
NOW+2MONTHS
```

This is one day ago:

```
NOW-1DAY
```

A slash is used to indicate rounding. This represents the beginning of the current hour:

```
NOW/HOUR
```

The following example computes (with millisecond precision) the point in time six months and three days into the future and then rounds that time to the beginning of that day:

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

Note that while date math is most commonly used relative to `NOW` it can be applied to any fixed moment in time as well:

```
1972-05-20T17:33:18.772Z+6MONTHS+3DAYS/DAY
```

#### *Request Parameters That Affect Date Math*

##### `NOW`

The `NOW` parameter is used internally by Solr to ensure consistent date math expression parsing across multiple nodes in a distributed request. But it can be specified to instruct Solr to use an arbitrary moment in time (past or future) to override for all situations where the the special value of "`NOW`" would impact date math expressions.

It must be specified as a (long valued) milliseconds since epoch

Example:

`q=solr&fq=start_date:[* TO NOW]&NOW=1384387200000`

#### `TZ`

By default, all date math expressions are evaluated relative to the UTC TimeZone, but the `TZ` parameter can be specified to override this behaviour, by forcing all date based addition and rounding to be relative to the specified time zone.

For example, the following request will use range faceting to facet over the current month, "per day" relative UTC:

```
http://localhost:8983/solr/select?q=*:*&facet.range=my_date_field&facet=true&facet.ran
ge.start=NOW/MONTH&facet.range.end=NOW/MONTH%2B1MONTH&facet.range.gap=%2B1DAY
```

```
<int name="2013-11-01T00:00:00Z">0</int>
<int name="2013-11-02T00:00:00Z">0</int>
<int name="2013-11-03T00:00:00Z">0</int>
<int name="2013-11-04T00:00:00Z">0</int>
<int name="2013-11-05T00:00:00Z">0</int>
<int name="2013-11-06T00:00:00Z">0</int>
<int name="2013-11-07T00:00:00Z">0</int>
...
```

While in this example, the "days" will be computed relative to the specified time zone - including any applicable Daylight Savings Time adjustments:

```
http://localhost:8983/solr/select?q=*:*&facet.range=my_date_field&facet=true&facet.ran
ge.start=NOW/MONTH&facet.range.end=NOW/MONTH%2B1MONTH&facet.range.gap=%2B1DAY&TZ=Ameri
ca/Los_Angeles
```

```
<int name="2013-11-01T07:00:00Z">0</int>
<int name="2013-11-02T07:00:00Z">0</int>
<int name="2013-11-03T07:00:00Z">0</int>
<int name="2013-11-04T08:00:00Z">0</int>
<int name="2013-11-05T08:00:00Z">0</int>
<int name="2013-11-06T08:00:00Z">0</int>
<int name="2013-11-07T08:00:00Z">0</int>
...
```

## Working with Enum Fields

The EnumField type allows defining a field whose values are a closed set, and the sort order is pre-determined but is not alphabetic nor numeric. Examples of this are severity lists, or risk definitions.

#### Defining an EnumField in `schema.xml`

The EnumField type definition is quite simple, as in this example defining field types for "priorityLevel" and "riskLevel" enumerations:

```
<fieldType name="priorityLevel" class="solr.EnumField" enumsConfig="enumsConfig.xml"
enumName="severity"/>
<fieldType name="riskLevel"     class="solr.EnumField" enumsConfig="enumsConfig.xml"
enumName="risk"    />
```

Besides the `name` and the `class`, which are common to all field types, this type also takes two additional parameters:

- `enumsConfig`: the name of a configuration file that contains the `<enum/>` list of field values and their order that you wish to use with this field type. If a path to the file is not defined specified, the file should be in the `conf` directory for the collection.
- `enumName`: the name of the specific enumeration in the `enumsConfig` file to use for this type.

#### Defining the EnumField configuration file

The file named with the `enumsConfig` parameter can contain multiple enumeration value lists with different names if there are multiple uses for enumerations in your Solr schema.

In this example, there are two value lists defined. Each list is between `enum` opening and closing tags:

```
<?xml version="1.0" ?>
<enumsConfig>
  <enum name="priority">
    <value>Not Available</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High"</value>
    <value>Urgent</value>
  </enum>
  <enum name="risk">
    <value>Unknown</value>
    <value>Very Low</value>
    <value>Low</value>
    <value>Medium</value>
    <value>High</value>
    <value>Critical</value>
  </enum>
</enumsConfig>
```

⚠ **Changing Values**
You cannot change the order, or remove, existing values in an `<enum/>` without reindexing.

You can however add new values to the end.

### Working with External Files and Processes

#### The `ExternalFileField` Type

The `ExternalFileField` type makes it possible to specify the values for a field in a file outside the Solr index. 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 for this field in the external file.

⚠ External fields are not searchable. They can be used only for function queries or display. For more

> information on function queries, see the section on Function Queries.

The `ExternalFileField` type 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 implemented a document rank based on the 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.

In `schema.xml`, the definition of this field type might look like this:

```
<fieldType name="entryRankFile" keyField="pkId" defVal="0" stored="false"
indexed="false" class="solr.ExternalFileField" valType="pfloat"/>
```

The `keyField` attribute defines the key that will be defined in the external file. It is usually the unique key for the index, but it doesn't need to be as long as the `keyField` can be used to identify documents in the index. A `defVal` defines a default value that will be used if there is no entry in the external file for a particular document.

The `valType` attribute specifies the actual type of values that will be found in the file. The type specified must be either a float field type, so valid values for this attribute are `pfloat`, `float` or `tfloat`. This attribute can be omitted.

### *Format of the External File*

The file itself is located in Solr's index directory, which by default is `$SOLR_HOME/data`. 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`.

> If any files using the name pattern `.*` (such as `.txt`) 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
```

The keys listed in this file do not need to be unique. The file does not need to be sorted, but Solr will be able to perform the lookup faster if it is.

### *Reloading an External File*

As of Solr 4.1, it's possible to define an event listener to reload an external file when either a searcher is reloaded or when a new searcher is started. See the section Query-Related Listeners for more information, but a sample definition in `solrconfig.xml` might look like this:

```
<listener event="newSearcher"
class="org.apache.solr.schema.ExternalFileFieldReloader"/>
<listener event="firstSearcher"
class="org.apache.solr.schema.ExternalFileFieldReloader"/>
```

#### Pre-Analyzing a Field Type

The `PreAnalyzedField` type provides a way to send to Solr serialized token streams, optionally with independent stored values of a field, and have this information stored and indexed without any additional text processing applied in Solr. This is useful if user wants to submit field content that was already processed by some existing external text processing pipeline (e.g., it has been tokenized, annotated, stemmed, synonyms inserted, etc.), while using all the rich attributes that Lucene's TokenStream provides (per-token attributes).

The serialization format is pluggable using implementations of PreAnalyzedParser interface. There are two out-of-the-box implementations:

- JsonPreAnalyzedParser: as the name suggests, it parses content that uses JSON to represent field's content. This is the default parser to use if the field type is not configured otherwise.
- SimplePreAnalyzedParser: uses a simple strict plain text format, which in some situations may be easier to create than JSON.

There is only one configuration parameter, `parserImpl`. The value of this parameter should be a fully qualified class name of a class that implements PreAnalyzedParser interface. The default value of this parameter is `org.apache.solr.schema.JsonPreAnalyzedParser`.

## Field Properties by Use Case

Here is a summary of common use cases, and the attributes the fields or field types should have to support the case. An entry of true or false in the table indicates that the option must be set to the given value for the use case to function correctly. If no entry is provided, the setting of that attribute has no impact on the case.

| Use Case | indexed | stored | multiValued | omitNorms | termVectors | termPositions | docValues |
|---|---|---|---|---|---|---|---|
| search within field | true | | | | | | |
| retrieve contents | | true | | | | | |
| use as unique key | true | | false | | | | |
| sort on field | true[7] | | false | true [1] | | | true[7] |
| use field boosts [5] | | | | false | | | |
| document boosts affect searches within field | | | | false | | | |
| highlighting | true [4] | true | | | true[2] | true [3] | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| faceting [5] | true[7] | | | | | | true[7] |
| add multiple values, maintaining order | | | true | | | | |
| field length affects doc score | | | | false | | | |
| MoreLikeThis [5] | | | | | true [6] | | |

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 Understanding Analyzers, Tokenizers, and Filters.

[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`.

[7] Either `indexed` or `docValues` must be true, but both are not required. DocValues can be more efficient in many cases.

## Defining Fields

Fields are defined in the fields element of `schema.xml`. Once you have the field types set up, defining the fields themselves is simple.

#### Example

The following example defines a field named `price` with a type named `float` and a default value of `0.0`; the `indexed` and `stored` properties are explicitly set to `true`, while any other properties specified on the `float` field type are inherited.

```
<field name="price" type="float" default="0.0" indexed="true" stored="true"/>
```

#### Field Properties

| Property | Description |
|---|---|
| name | The name of the field.  Field names should consist of alphanumeric or underscore characters only and not start with a digit. This is not currently strictly enforced, but other field names will not have first class support from all components and back compatibility is not guaranteed. Names with both leading and trailing underscores (e.g. `_version_`) are reserved. Every field must have a `name`. |
| type | The name of the `fieldType` for this field.  This will be found in the "`name`" attribute on the `fieldType` definition.  Every field must have a `type`. |
| default | A default value that will be added automatically to any document that does not have a value in this field when it is indexed.  If this property is not specified, there is no default. |

### Optional Field Type Override Properties

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. Included below is the table of field type properties from the section Field Type Definitions and Properties:

| 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 |
| docValues | If true, the value of the field will be put in a column-oriented DocValues str ucture | true or false |
| sortMissingFirst<br>sortMissingLast | Control the placement of documents when a sort field is not present. As of Solr 3.5, these work for all numeric fields, including Trie and date fields. | true or false |
| multiValued | If true, indicates that a single document might contain multiple values for this field type | true or false |
| 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). Defaults to true for all primitive (non-analyzed) field types, such as int, float, data, bool, and string. 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 |
| omitPositions | Similar to `omitTermFreqAndPositions` but preserves term frequency information | true or false |
| termVectors<br>termPositions<br>termOffsets | These options instruct Solr to maintain full term vectors for each document, optionally including the position and offset information for each term occurrence in those vectors. These can be used to accelerate highlighting and other ancillary functionality, but impose a substantial cost in terms of index size. They are not necessary for typical uses of Solr | true or false |
| required | Instructs Solr to reject any attempts to add a document which does not have a value for this field. This property defaults to false. | true or false |

### Related Topics

- SchemaXML-Fields
- Field Options by Use Case

## 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.

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` destination field has data of its own in the input documents, the contents of the `cat` field will be added as additional values – just as if all of the values had originally been specified by the client.  Remember to configure your fields as `multivalued="true"` if they will ultimately get multiple values (either from a multivalued source, or multiple `copyField` directives, etc...)

The `maxChars` parameter, an `int` parameter, establishes an upper limit for the number of characters to be copied from the source value when constructing the value added to the destination field. This limit is useful for situations in which you want to copy some data from the source field, but also control the size of index files.

Both the source and the destination of `copyField` can contain either leading or trailing 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" />
```

> ⚠ 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.

**Related Topics**

- SchemaXML-Copy Fields

## Dynamic Fields

*Dynamic fields* allow Solr to index fields that you did not explicitly define in your schema. This is useful 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="int" indexed="true"  stored="true"/>
```

It is recommended that you include basic dynamic field mappings (like that shown above) in your `schema.xml`. The mappings can be very useful.

### Related Topics

- SchemaXML-Dynamic Fields

## Other Schema Elements

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

### 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.

You can define the unique key field by naming it:

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

Starting with Solr 4, schema defaults and `copyFields` cannot be used to populate the `uniqueKey` field. You also can't use `UUIDUpdateProcessorFactory` to have `uniqueKey` values generated automatically.

Further, the operation will fail if the `uniqueKey` field is used, but is multivalued (or inherits the multivalueness from the `fieldtype`). However, `uniqueKey` will continue to work, as long as the field is properly used.

### 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 and Extended DisMax query parsers do not use this value.

> Use of the `defaultSearchField` element is deprecated in Solr versions 3.6 and higher. Instead, you should use the `df` request parameter. At some point, the `defaultSearchField` element may be removed.

For more information about query parsers, see the section on Query Syntax and Parsing.

### Query Parser Default 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`, the `solrQueryParser` element controls 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 or Extended DisMax query parsers, which internally hard-code their operators to OR.

> The query parser default operator parameter has been deprecated in Solr versions 3.6 and higher. You are instead encouraged to specify the query parser `q.op` parameter in your request handler.

### Similarity

Similarity is a Lucene class used to score a document in searching. This class can be changed in order to provide a more custom sorting. With Solr 4, you can configure a different `similarity` for each field, meaning that scoring a document will differ depending on what's in each field. However, you can still configure a global `similarity` is configured in the schema.xml file, where an implicit instance of `DefaultSimilarityFactory` is used.

A global `<similarity>` declaration can be used to specify a custom similarity implementation that you want Solr to use when dealing with your index. A similarity can be specified either by referring directly to the name of a class with a no-argument constructor:

```
<similarity class="solr.DefaultSimilarityFactory"/>
```

or by referencing a `SimilarityFactory` implementation, which may take optional initialization parameters:

```
<similarity class="solr.DFRSimilarityFactory">
  <str name="basicModel">P</str>
  <str name="afterEffect">L</str>
  <str name="normalization">H2</str>
  <float name="c">7</float>
</similarity>
```

Beginning with Solr 4, similarity factories can be specified on individual field types:

```
<fieldType name="text_ib">
   <analyzer/>
   <similarity class="solr.IBSimilarityFactory">
      <str name="distribution">SPL</str>
      <str name="lambda">DF</str>
      <str name="normalization">H2</str>
   </similarity>
</fieldType>
```

This example uses `IBSimilarityFactory` (using the Information-Based model), but there are several similarity implementations that can be used. For Solr 4.2, `SweetSpotSimilarityFactory` has been added. Other options include `BM25SimilarityFactory`, `DFRSimilarityFactory`, `SchemaSimilarityFactory` and others. For details, see the Solr Javadocs for the similarity factories.

### Related Topics

- SchemaXML-Miscellaneous Settings
- UniqueKey

## Schema API

The Solr schema API allows using a REST API to get information about the `schema.xml` for each collection (or core for standalone Solr), including defined field types, fields, dynamic fields, and copy field declarations. In Solr 4.2 and 4.3, it only allows GET (read-only) access, but in Solr 4.4, new fields and copyField directives may be added to the schema. Future Solr releases will extend this functionality to allow more schema elements to be updated.

To enable schema modification with this API, the schema will need to be managed and mutable. See the section Managed Schema Definition in SolrConfig for more information.

The API allows two output modes for all calls: JSON or XML. When requesting the complete schema, there is another output mode which is XML modeled after the schema.xml file itself.

The base address for the API is `http://<host>:<port>/<context-path>`, where `<context-path>` is usually `solr`, though you may have configured it differently. Example base address: `http://localhost:8983/solr` .

In the API entry points and example URLs below, you may alternatively specify a Solr *core* name where it says *collection*.

### API Entry Points

`/collection/schema`: retrieve the entire schema

`/collection/schema/fields`: retrieve information about all defined fields, or create new fields with optional copyField directives

`/collection/schema/fields/name` : retrieve information about a named field, or create a new named field with optional copyField directives

`/collection/schema/dynamicfields`: retrieve information about dynamic field rules

`/collection/schema/dynamicfields/name` : retrieve information about a named dynamic rule

`/collection/schema/fieldtypes`: retrieve information about field types

`/collection/schema/fieldtypes/name` : retrieve information about a named field type

`/collection/schema/copyfields`: retrieve information about copy fields, or create new copyField directives

`/collection/schema/name`: retrieve the schema name

`/collection/schema/version`: retrieve the schema version

`/collection/schema/uniquekey`: retrieve the defined uniqueKey

`/collection/schema/similarity`: retrieve the global similarity definition

`/collection/schema/solrqueryparser/defaultoperator`: retrieve the default operator

`/collection/schema/managed/resource/paths`: Manipulate managed resource data

### Retrieve schema information

#### Retrieve the Entire Schema

GET */collection/*schema

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json**, **xml** or **schema.xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include all fields, field types, dynamic rules and copy field rules. The schema name and version are also included.

## Examples

### Input

Get the entire schema in JSON.

```
curl http://localhost:8983/solr/collection1/schema?wt=json
```

Get the entire schema in XML.

```
curl http://localhost:8983/solr/collection1/schema?wt=xml
```

Get the entire schema in "schema.xml" format.

```
curl http://localhost:8983/solr/collection1/schema?wt=schema.xml
```

### Output

The samples below have been truncated to only show a few snippets of the output.

Example output in JSON:

```
{
  "responseHeader":{
    "status":0,
    "QTime":5},
  "schema":{
    "name":"example",
    "version":1.5,
    "uniqueKey":"id",
    "fieldTypes":[{
        "name":"alphaOnlySort",
        "class":"solr.TextField",
        "sortMissingLast":true,
        "omitNorms":true,
        "analyzer":{
          "tokenizer":{
            "class":"solr.KeywordTokenizerFactory"},
          "filters":[{
              "class":"solr.LowerCaseFilterFactory"},
            {
              "class":"solr.TrimFilterFactory"},
            {
              "class":"solr.PatternReplaceFilterFactory",
              "replace":"all",
              "replacement":"",
              "pattern":"([^a-z])"}]}},
...
    "fields":[{
        "name":"_version_",
        "type":"long",
        "indexed":true,
        "stored":true},
      {
        "name":"author",
        "type":"text_general",
        "indexed":true,
        "stored":true},
      {
        "name":"cat",
        "type":"string",
        "multiValued":true,
        "indexed":true,
        "stored":true},
...
    "copyFields":[{
        "source":"author",
        "dest":"text"},
      {
        "source":"cat",
        "dest":"text"},
      {
        "source":"content",
        "dest":"text"},
...
      {
        "source":"author",
        "dest":"author_s"}]}}
```

Example output in XML:

```
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">5</int>
</lst>
<lst name="schema">
  <str name="name">example</str>
  <float name="version">1.5</float>
  <str name="uniqueKey">id</str>
  <arr name="fieldTypes">
    <lst>
      <str name="name">alphaOnlySort</str>
      <str name="class">solr.TextField</str>
      <bool name="sortMissingLast">true</bool>
      <bool name="omitNorms">true</bool>
      <lst name="analyzer">
        <lst name="tokenizer">
          <str name="class">solr.KeywordTokenizerFactory</str>
        </lst>
        <arr name="filters">
          <lst>
            <str name="class">solr.LowerCaseFilterFactory</str>
          </lst>
          <lst>
            <str name="class">solr.TrimFilterFactory</str>
          </lst>
          <lst>
            <str name="class">solr.PatternReplaceFilterFactory</str>
            <str name="replace">all</str>
            <str name="replacement"/>
            <str name="pattern">([^a-z])</str>
          </lst>
        </arr>
      </lst>
    </lst>
...
    <lst>
      <str name="source">author</str>
      <str name="dest">author_s</str>
    </lst>
  </arr>
</lst>
</response>
```

Example output in schema.xml format:

```
<schema name="example" version="1.5">
  <uniqueKey>id</uniqueKey>
  <types>
    <fieldType name="alphaOnlySort" class="solr.TextField" sortMissingLast="true"
omitNorms="true">
      <analyzer>
        <tokenizer class="solr.KeywordTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.TrimFilterFactory"/>
        <filter class="solr.PatternReplaceFilterFactory" replace="all" replacement=""
pattern="([^a-z])"/>
      </analyzer>
    </fieldType>
...
  <copyField source="url" dest="text"/>
  <copyField source="price" dest="price_c"/>
  <copyField source="author" dest="author_s"/>
</schema>
```

#### List Fields

`GET /collection/schema/fields`

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include each field and any defined configuration for each field. The defined configuration can vary for each field, but will minimally include the field `name`, the `type`, if it is `indexed` and if it is `stored`. If `multiValued` is defined as either true or false (most likely true), that will also be shown. See the section Defining Fields for more information about each parameter.

## Examples

### Input

Get a list of all fields.

```
curl http://localhost:8983/solr/collection1/schema/fields?wt=json
```

**Output**

The sample output below has been truncated to only show a few fields.

```
{
    "fields": [
        {
            "indexed": true,
            "name": "_version_",
            "stored": true,
            "type": "long"
        },
        {
            "indexed": true,
            "name": "author",
            "stored": true,
            "type": "text_general"
        },
        {
            "indexed": true,
            "multiValued": true,
            "name": "cat",
            "stored": true,
            "type": "string"
        },
...
    ],
    "responseHeader": {
        "QTime": 1,
        "status": 0
    }
}
```

#### List a Specific Field

GET /*collection*/schema/fields/*fieldname*

### Input

#### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |
| fieldname | The specific field name. |

#### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include each field and any defined configuration for the field. The defined configuration can vary for a field, but will minimally include the field `name`, the `type`, if it is `indexed` and if it is `stored`. If `multiValued` is defined as either true or false (most likely true), that will also be shown. See the section Defining Fields for more information about each parameter.

## Examples

### Input

Get the 'author' field.

```
curl http://localhost:8983/solr/collection1/schema/fields/author?wt=json
```

### Output

```
{
    "field": {
        "indexed": true,
        "name": "author",
        "stored": true,
        "type": "text_general"
    },
    "responseHeader": {
        "QTime": 2,
        "status": 0
    }
}
```

### List Dynamic Fields

`GET /`*`collection`*`/schema/dynamicfields`

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include each dynamic field rule and the defined configuration for each rule. The defined configuration can vary for each rule, but will minimally include the dynamic field `name`, the `type`, if it is `indexed` and if it is `stored`. See the section Dynamic Fields for more information about each parameter.

## Examples

### Input

Get a list of all dynamic field declarations

```
curl http://localhost:8983/solr/collection1/schema/dynamicfields?wt=json
```

### Output

The sample output below has been truncated.

```
{
    "dynamicFields": [
        {
            "indexed": true,
            "name": "*_coordinate",
            "stored": false,
            "type": "tdouble"
        },
        {
            "multiValued": true,
            "name": "ignored_*",
            "type": "ignored"
        },
        {
            "name": "random_*",
            "type": "random"
        },
        {
            "indexed": true,
            "multiValued": true,
            "name": "attr_*",
            "stored": true,
            "type": "text_general"
        },
        {
            "indexed": true,
            "multiValued": true,
            "name": "*_txt",
            "stored": true,
            "type": "text_general"
        }
...
    ],
    "responseHeader": {
        "QTime": 1,
        "status": 0
    }
}
```

### List a Specific Dynamic Field Rule

GET */collection/*schema/dynamicfields/*name*

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |
| name | The name of the dynamic field rule. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include the requested dynamic field rule and any defined configuration for the rule. The defined configuration can vary for each rule, but will minimally include the dynamic field `name`, the `type`, if it is `indexed` and if it is `stored`. See the section Dynamic Fields for more information about each parameter.

## Examples

### Input

Get the details of the "*_s" rule.

```
curl http://localhost:8983/solr/collection1/schema/dynamicfields/*_s?wt=json
```

### Output

```
{
    "dynamicfield": {
        "indexed": true,
        "name": "*_s",
        "stored": true,
        "type": "string"
    },
    "responseHeader": {
        "QTime": 1,
        "status": 0
    }
}
```

### List Field Types

```
GET /collection/schema/fieldtypes
```

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include each field type and any defined configuration for the type. The defined configuration can vary for each type, but will minimally include the field type `name` and the `class`. If query or index analyzers, tokenizers, or filters are defined, those will also be shown with other defined parameters. See the section Solr Field Types for more information about how to configure various types of fields.

## Examples

### Input

Get a list of all field types.

```
curl http://localhost:8983/solr/collection1/schema/fieldtypes?wt=json
```

### Output

The sample output below has been truncated to show a few different field types from different parts of the list.

```
{
    "fieldTypes": [
        {
            "analyzer": {
                "class": "solr.TokenizerChain",
                "filters": [
                    {
                        "class": "solr.LowerCaseFilterFactory"
                    },
                    {
                        "class": "solr.TrimFilterFactory"
                    },
                    {
                        "class": "solr.PatternReplaceFilterFactory",
                        "pattern": "([^a-z])",
                        "replace": "all",
                        "replacement": ""
                    }
                ],
                "tokenizer": {
                    "class": "solr.KeywordTokenizerFactory"
                }
            },
            "class": "solr.TextField",
            "dynamicFields": [],
            "fields": [],
            "name": "alphaOnlySort",
            "omitNorms": true,
            "sortMissingLast": true
        },
...
        {
            "class": "solr.TrieFloatField",
            "dynamicFields": [
                "*_fs",
                "*_f"
            ],
            "fields": [
                "price",
                "weight"
            ],
            "name": "float",
            "positionIncrementGap": "0",
            "precisionStep": "0"
        },
...
}
```

#### List a Specific Field Type

`GET` *`/collection/`*`schema/fieldtypes/`*`name`*

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |
| name | The name of the field type. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include each field type and any defined configuration for the type. The defined configuration can vary for each type, but will minimally include the field type `name` and the `class`. If query and/or index analyzers, tokenizers, or filters are defined, those will be shown with other defined parameters. See the section Solr Field Types for more information about how to configure various types of fields.

## Examples

### Input

Get details of the "date" field type.

```
curl http://localhost:8983/solr/collection1/schema/fieldtypes/date?wt=json
```

### Output

The sample output below has been truncated.

```
{
    "fieldType": {
        "class": "solr.TrieDateField",
        "dynamicFields": [
            "*_dts",
            "*_dt"
        ],
        "fields": [
            "last_modified"
        ],
        "name": "date",
        "positionIncrementGap": "0",
        "precisionStep": "0"
    },
    "responseHeader": {
        "QTime": 2,
        "status": 0
    }
}
```

### List Copy Fields

GET /*collection*/schema/copyfields

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include the `source` and `destination` of each copy field rule defined in `schema.xml`. For more information about copying fields, see the section Copying Fields.

## Examples

### Input

Get a list of all copyfields.

```
curl http://localhost:8983/solr/collection1/schema/copyfields?wt=json
```

### Output

The sample output below has been truncated to the first few copy definitions.

```
{
    "copyFields": [
        {
            "dest": "text",
            "source": "author"
        },
        {
            "dest": "text",
            "source": "cat"
        },
        {
            "dest": "text",
            "source": "content"
        },
        {
            "dest": "text",
            "source": "content_type"
        },
...
    ],
    "responseHeader": {
        "QTime": 3,
        "status": 0
    }
}
```

#### Show Schema Name

`GET /`*`collection`*`/schema/name`

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will be simply the name given to the schema.

## Examples

### Input

Get the schema name.

```
curl http://localhost:8983/solr/collection1/schema/name?wt=json
```

#### Output

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "name":"example"}
```

### Show the Schema Version

GET */collection/*schema/version

#### Input

#### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

#### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

#### Output

#### Output Content

The output will simply be the schema version in use.

#### Examples

#### Input

Get the schema version

```
curl http://localhost:8983/solr/collection1/schema/version?wt=json
```

#### Output

```
{
  "responseHeader":{
    "status":0,
    "QTime":2},
  "version":1.5}
```

### List UniqueKey

GET */collection/*schema/uniquekey

### Input

#### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

#### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

### Output

#### Output Content

The output will include simply the field name that is defined as the uniqueKey for the index.

### Examples

#### Input

List the uniqueKey.

```
curl http://localhost:8983/solr/collection1/schema/uniquekey?wt=json
```

#### Output

The sample output below has been truncated to the first few copy definitions.

```
{
  "responseHeader":{
    "status":0,
    "QTime":2},
  "uniqueKey":"id"}
```

## Show Global Similarity

GET */collection/*schema/similarity

### Input

#### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

#### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include the class name of the global similarity defined (if any).

## Examples

### Input

Get the similarity implementation.

```
curl http://localhost:8983/solr/collection1/schema/similarity?wt=json
```

### Output

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "similarity":{
    "class":"org.apache.solr.search.similarities.DefaultSimilarityFactory"}}
```

### Get the Default Query Operator

GET */collection/*schema/solrqueryparser/defaultoperator

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, JSON will be returned by default. |

## Output

### Output Content

The output will include simply the default operator if none is defined by the user.

### Examples

#### Input

Get the default operator.

```
curl
http://localhost:8983/solr/collection1/schema/solrqueryparser/defaultoperator?wt=json
```

#### Output

```
{
  "responseHeader":{
    "status":0,
    "QTime":2},
  "defaultOperator":"OR"}
```

## Modify the schema

### Create new schema fields

POST /*collection*/schema/fields

To enable schema modification, the schema will need to be managed and mutable. See the section Managed Schema Definition in SolrConfig for more information.

### Input

#### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

#### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, json will be returned by default. |

#### Request body

Only JSON format is supported in the request body. The JSON must contain an array of one or more new field specifications, each of which must include mappings for the new field's `name` and `type`. All attributes specifiable on a schema `<field name="..." ... />` declaration may be specified here - see Defining Fields.

Additionally, `copyField` destination(s) may optionally be specified. Note that each specified copyField destination must be an existing schema field (and not a dynamic field). In particular, since the new fields specified in a new field creation request are defined all at once, you cannot specify a `copyField` that targets another new field in the same request - instead, you have to make two requests, defining the `copyField` destination in the first new field creation request, then specifying that field as a `copyField` destination in the second new field creation request.

The `curl` utility can provide the request body via its `--data-binary` option.

## Output

### Output Content

The output will be the response header, containing a status code, and if there was a problem, an associated error message.

Example output in the default JSON format:

```
{
  "responseHeader":{
    "status":0,
    "QTime":8}}
```

## Examples

### Input

Add two new fields:

```
curl http://localhost:8983/solr/collection1/schema/fields -X POST -H
'Content-type:application/json' --data-binary '
[
    {
        "name":"sell-by",
        "type":"tdate",
        "stored":true
    },
    {
        "name":"catchall",
        "type":"text_general",
        "stored":false
    }
]'
```

Add a third new field and copy it to the "catchall" field created above:

```
curl http://localhost:8983/solr/collection1/schema/fields -X POST -H
'Content-type:application/json' --data-binary '
[
    {
        "name":"department",
        "type":"string",
        "docValues":"true",
        "default":"no department",
        "copyFields": [ "catchall" ]
    }
]'
```

### Create one new schema field

PUT */collection*/schema/fields/*name*

To enable schema modification, the schema will need to be managed and mutable. See the section Managed

Schema Definition in SolrConfig for more information.

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |
| name | The new field name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, json will be returned by default. |

### Request body

Only JSON format is supported in the request body. The body must include a set of mappings, minimally for the new field's `name` and `type`. All attributes specifiable on a schema `<field name="..." ... />` declaration may be specified here - see Defining Fields.

Additionally, `copyField` destination(s) may optionally be specified. Note that each specified copyField destination must be an existing schema field (and not a dynamic field).

The `curl` utility can provide the request body via its `--data-binary` option.

## Output

### Output Content

The output will be the response header, containing a status code, and if there was a problem, an associated error message.

Example output in the default JSON format:

```
{
  "responseHeader":{
    "status":0,
    "QTime":4}}
```

## Examples

### Input

Add a new field named "narrative":

```
curl http://localhost:8983/solr/collection1/schema/fields/narrative -X PUT -H
'Content-type:application/json' --data-binary '
{
    "type":"text_general",
    "stored":true,
    "termVectors":true,
    "termPositions":true,
    "termOffsets":true
}'
```

Add a new field named "color" and copy it to two fields, named "narrative" and "catchall", which must already exist in the schema:

```
curl http://localhost:8983/solr/collection1/schema/fields/color -X PUT -H
'Content-type:application/json' --data-binary '
{
    "type":"string",
    "stored":true,
    "copyFields": [
        "narrative",
        "catchall"
    ]
}'
```

#### Create new copyField directives

`POST` */collection*`/schema/copyfields`

To enable schema modification, the schema will need to be managed and mutable. See the section Managed Schema Definition in SolrConfig for more information.

## Input

### Path Parameters

| Key | Description |
|---|---|
| collection | The collection (or core) name. |

### Query Parameters

The query parameters can be added to the API request after a '?'.

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| wt | string | No | json | Defines the format of the response. The options are **json** or **xml**. If not specified, json will be returned by default. |

### Request body

Only JSON format is supported in the request body. The body must contain an array of zero or more copyField directives, each containing a mapping from `source` to the source field name, and from `dest` to an array of destination field name(s).

`source` field names must either be an existing field, or be a field name glob (with an asterisk either at the beginning or the end, or consist entirely of a single asterisk). `dest` field names must either be existing fields, or, if `source` is a glob, `dest` fields may be globs that match an existing dynamic field.

The `curl` utility can provide the request body via its `--data-binary` option.

## Output

### Output Content

The output will be the response header, containing a status code, and if there was a problem, an associated error message.

Example output in the default JSON format:

```
{
  "responseHeader":{
    "status":0,
    "QTime":2}}
```

## Examples

### Input

Copy the "affiliations" field to the "relations" field, and the "shelf" field to the "location" and "catchall" fields:

```
curl http://localhost:8983/solr/collection1/schema/copyfields -X POST -H
'Content-type:application/json' --data-binary '
[
    {
        "source":"affiliations",
        "dest": [
            "relations"
        ]
    },
    {
        "source":"shelf",
        "dest": [
            "location",
            "catchall"
        ]
    }
]'
```

Copy all fields names matching "finance_*" to the "*_s" dynamic field:

```
curl http://localhost:8983/solr/collection1/schema/copyfields -X POST -H
'Content-type:application/json' --data-binary '
[
    {
        "source":"finance_*",
        "dest": [
            "*_s"
        ]
    }
]'
```

#### Manage Resource Data

The Managed Resources REST API provides a mechanism for any Solr plugin to expose resources that should support CRUD (Create, Read, Update, Delete) operations.  Depending on what Field Types and Analyzers are configured in your Schema, additional `/schema/` REST API paths may exist.  See the Managed Resources section for more information and examples.

### Related Topics

- Managed Schema Definition in SolrConfig

## 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 structure 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.

### Choosing Appropriate Numeric Types

For general numeric needs, use `TrieIntField`, `TrieLongField`, `TrieFloatField`, and `TrieDoubleField` with `precisionStep="0"`.

If you expect users to make frequent range queries on numeric types, use the default `precisionStep` (by not specifying it) or specify it as `precisionStep="8"` (which is the default). This offers faster speed for range queries at the expense of increasing index size.

### 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`

### Related Topics

- SchemaXML

## DocValues

An exciting addition to Solr functionality was introduced in Solr 4.2. This functionality has been around in Lucene for a while, but is now available to Solr users.

DocValues are a way of building the index that is more efficient for some purposes.

### Why DocValues?

The standard way that Solr builds the index is with an *inverted index*. This style builds a list of terms found in all the documents in the index and next to each term is a list of documents that the term appears in (as well as how many times the term appears in that document). This makes search very fast - since users search by terms, having a ready list of term-to-document values makes the query process faster.

For other features that we now commonly associate with search, such as sorting, faceting, and highlighting, this approach is not very efficient. The faceting engine, for example, must look up each term that appears in each document that will make up the result set and pull the document IDs in order to build the facet list. In Solr, this is maintained in memory, and can be slow to load (depending on the number of documents, terms, etc.).

In Lucene 4.0, a new approach was introduced. DocValue fields are now column-oriented fields with a document-to-value mapping built at index time. This approach promises to relieve some of the memory requirements of the fieldCache and make lookups for faceting, sorting, and grouping much faster.

### How to Use DocValues

To use docValues, you only need to enable it for a field that you will use it with. As with all schema design, you need to define a field type and then define fields of that type with docValues enabled. All of these actions are done in `schema.xml`.

Enabling a field for docValues only requires adding `docValues="true"` to the field definition, as in this example (from Solr's default `schema.xml`):

```
<field name="manu_exact" type="string" indexed="false" stored="false" docValues="true"
/>
```

Prior to Solr 4.5, a field could not be empty to be used with docValues; in Solr 4.5, that restriction is removed.

> If you have already indexed data into your Solr index, you will need to completely re-index your content after changing your field definitions in `schema.xml` in order to successfully use docValues.

DocValues are only available for specific field types. The types chosen determine the underlying Lucene docValue type that will be used. The available Solr field types are:

- String fields of type `StrField`.
  - If the field is single-valued (i.e., multi-valued is false), Lucene will use the SORTED type.
  - If the field is multi-valued, Lucene will use the SORTED_SET type.
- Any Trie* fields.
  - If the field is single-valued (i.e., multi-valued is false), Lucene will use the NUMERIC type.
  - If the field is multi-valued, Lucene will use the SORTED_SET type.
- UUID fields

These Lucene types are related to how the values are sorted and stored.

There is an additional configuration option available, which is to modify the `docValuesFormat` used by the field type. The default implementation employs a mixture of loading some things into memory and keeping some on disk. In some cases, however, you may choose to specify an alternative DocValuesFormat implementation. For example, you could choose to keep everything in memory by specifying `docValuesFormat="Memory"` on a field type:

```
<fieldType name="string_in_mem_dv" class="solr.StrField" docValues="true"
docValuesFormat="Memory" />
```

Please note that the `docValuesFormat` option may change in future releases.

> Lucene index back-compatibility is only supported for the default codec. If you choose to customize the `docValuesFormat` in your schema.xml, upgrading to a future version of Solr may require you to either switch back to the default codec and optimize your index to rewrite it into the default codec before upgrading, or re-build your entire index from scratch after upgrading.

### Related Topics

DocValues are quite new to Solr. For more background see:

- Introducing Lucene Index Doc Values, by Simon Willnauer, at SearchWorkings.org
- Fun with DocValues in Solr 4.2, by David Arthur, at SearchHub.org
- The old wiki page on DocValues (note, that page is now obsoleted by this one)

## Schemaless Mode

Schemaless Mode is a set of Solr features that, when used together, allow users to rapidly construct an effective schema by simply indexing sample data, without having to manually edit the schema. These Solr features, all specified in `solrconfig.xml`, are:

1. Managed schema: Schema modifications are made through Solr APIs rather than manual edits - see Managed Schema Definition in SolrConfig.
2. Field value class guessing: Previously unseen fields are run through a cascading set of value-based parsers, which guess the Java class of field values - parsers for Boolean, Integer, Long, Float, Double, and Date are currently available.
3. Automatic schema field addition, based on field value class(es): Previously unseen fields are added to the schema, based on field value Java classes, which are mapped to schema field types - see Solr Field Types.

### Using the Schemaless Example

The three features of schemaless mode are pre-configured in the `example/example-schemaless/solr/` directory in the Solr distribution. To start Solr in this pre-configured schemaless mode, go to the `example/` directory and start up Solr, setting the `solr.solr.`home system property to this directory on the command line:

```
java -Dsolr.solr.home=example-schemaless/solr -jar start.jar
```

The schema in `example-schemaless/solr/collection1/conf/` is shipped with only two fields, `id` and `_version_`, as can be seen from calling the `/schema/fields` Schema API - `curl http://localhost:8983/solr/schema/fields` outputs:

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "fields":[{
      "name":"_version_",
      "type":"long",
      "indexed":true,
      "stored":true},
    {
      "name":"id",
      "type":"string",
      "multiValued":false,
      "indexed":true,
      "required":true,
      "stored":true,
      "uniqueKey":true}]}
```

## Configuring Schemaless Mode

As described above, there are three configuration elements that need to be in place to use Solr in schemaless mode. If you use the `solrconfig.xml` from `example/example-schemaless` these elements are configured already. If, however, you would like to implement schemaless on your own, you should make the following changes.

### Enable Managed Schema

As described in the section Managed Schema Definition in SolrConfig, changing the `schemaFactory` will allow the schema to be modified by the Schema API. Your `solrconfig.xml` should have a section like the one below (and the ClassicIndexSchemaFactory should be commented out or removed).

```
   <schemaFactory class="ManagedIndexSchemaFactory">
    <bool name="mutable">true</bool>
    <str name="managedSchemaResourceName">managed-schema</str>
  </schemaFactory>
```

### Define an UpdateRequestProcessorChain

The UpdateRequestProcessorChain allows Solr to guess field types, and you can define the default field type classes to use. To start, you should define it as follows (see the javadoc links below for update processor factory documentation):

```
<updateRequestProcessorChain name="add-unknown-fields-to-the-schema">
  <!-- UUIDUpdateProcessorFactory will generate an id if none is present in the
incoming document -->
  <processor class="solr.UUIDUpdateProcessorFactory" />
  <processor class="solr.LogUpdateProcessorFactory"/>
  <processor class="solr.DistributedUpdateProcessorFactory"/>
  <processor class="solr.RemoveBlankFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseBooleanFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseLongFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseDoubleFieldUpdateProcessorFactory"/>
  <processor class="solr.ParseDateFieldUpdateProcessorFactory">
    <arr name="format">
      <str>yyyy-MM-dd'T'HH:mm:ss.SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss,SSSZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss.SSS</str>
      <str>yyyy-MM-dd'T'HH:mm:ss,SSS</str>
      <str>yyyy-MM-dd'T'HH:mm:ssZ</str>
      <str>yyyy-MM-dd'T'HH:mm:ss</str>
      <str>yyyy-MM-dd'T'HH:mmZ</str>
      <str>yyyy-MM-dd'T'HH:mm</str>
      <str>yyyy-MM-dd HH:mm:ss.SSSZ</str>
      <str>yyyy-MM-dd HH:mm:ss,SSSZ</str>
      <str>yyyy-MM-dd HH:mm:ss.SSS</str>
      <str>yyyy-MM-dd HH:mm:ss,SSS</str>
      <str>yyyy-MM-dd HH:mm:ssZ</str>
      <str>yyyy-MM-dd HH:mm:ss</str>
      <str>yyyy-MM-dd HH:mmZ</str>
      <str>yyyy-MM-dd HH:mm</str>
      <str>yyyy-MM-dd</str>
    </arr>
  </processor>
  <processor class="solr.AddSchemaFieldsUpdateProcessorFactory">
    <str name="defaultFieldType">text_general</str>
    <lst name="typeMapping">
      <str name="valueClass">java.lang.Boolean</str>
      <str name="fieldType">booleans</str>
    </lst>
    <lst name="typeMapping">
      <str name="valueClass">java.util.Date</str>
      <str name="fieldType">tdates</str>
    </lst>
    <lst name="typeMapping">
      <str name="valueClass">java.lang.Long</str>
      <str name="valueClass">java.lang.Integer</str>
      <str name="fieldType">tlongs</str>
    </lst>
    <lst name="typeMapping">
      <str name="valueClass">java.lang.Number</str>
      <str name="fieldType">tdoubles</str>
    </lst>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

Javadocs for update processor factories mentioned above:

- UUIDUpdateProcessorFactory

- ParseBooleanFieldUpdateProcessorFactory
- ParseLongFieldUpdateProcessorFactory
- ParseDoubleFieldUpdateProcessorFactory
- ParseDateFieldUpdateProcessorFactory
- AddSchemaFieldsUpdateProcessorFactory

#### Make the UpdateRequestProcessorChain the Default for the UpdateRequestHandler

Once the UpdateRequestProcessorChain has been defined, you must instruct your UpdateRequestHandler to use it when working with index updates (i.e., adding, removing, replacing documents). Here is an example using the `/update` requestHandler:

```
<requestHandler name="/update" class="solr.UpdateRequestHandler">
    <lst name="defaults">
      <str name="update.chain">add-unknown-fields-to-the-schema</str>
    </lst>
  </requestHandler>
```

> After each of these changes have been made, Solr should be restarted (or, you can reload the cores to load the new `solrconfig.xml` definitions).

### Examples of Indexed Documents

Once the schemaless mode has been enabled (whether you configured it manually or are using `example-schemaless`), documents that include fields that are not defined in your schema should be added to the index, and the new fields added to the schema.

For example, adding a CSV document will cause its fields that are not in the schema to be added, with fieldTypes based on values:

```
curl "http://localhost:8983/solr/update?commit=true" -H "Content-type:application/csv"
-d '
id,Artist,Album,Released,Rating,FromDistributor,Sold
44C,Old Shews,Mead for Walking,1988-08-13,0.01,14,0'
```

Output indicating success:

```
<response>
  <lst name="responseHeader"><int name="status">0</int><int
name="QTime">106</int></lst>
</response>
```

The fields now in the schema (output from `curl http://localhost:8983/solr/schema/fields` ):

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "fields":[{
      "name":"Album",
      "type":"text_general"}, // Field value guessed as String -> text_general
fieldType
    {
      "name":"Artist",
      "type":"text_general"}, // Field value guessed as String -> text_general
fieldType
    {
      "name":"FromDistributor",
      "type":"tlongs"},       // Field value guessed as Long -> tlongs fieldType
    {
      "name":"Rating",
      "type":"tdoubles"},     // Field value guessed as Double -> tdoubles fieldType
    {
      "name":"Released",
      "type":"tdates"},       // Field value guessed as Date -> tdates fieldType
    {
      "name":"Sold",
      "type":"tlongs"},       // Field value guessed as Long -> tlongs fieldType
    {
      "name":"_version_",
...
    },
    {
      "name":"id",
...
    }]}
```

**You Can Still Be Explicit**
Even if you want to use schemaless mode for most fields, you can still use the Schema API to pre-emptively create some fields, with explicit types, before you index documents that use them.

Internally, the Schema REST API and the Schemaless Update Processors both use the same Managed Schema functionality.

Once a field has been added to the schema, its field type is fixed. As a consequence, adding documents with field value(s) that conflict with the previously guessed field type will fail. For example, after adding the above document, the `Sold` field has the fieldType `tlongs`, but the document below has a non-integral decimal value in this field:

```
curl "http://localhost:8983/solr/update?commit=true" -H "Content-type:application/csv"
-d '
id,Description,Sold
19F,Cassettes by the pound,4.93'
```

This document will fail, as shown in this output:

```
<response>
  <lst name="responseHeader">
    <int name="status">400</int>
    <int name="QTime">7</int>
  </lst>
  <lst name="error">
    <str name="msg">ERROR: [doc=19F] Error adding field 'Sold'='4.93' msg=For input
string: "4.93"</str>
    <int name="code">400</int>
  </lst>
</response>
```

# Understanding Analyzers, Tokenizers, and Filters

The following sections describe how Solr breaks down and works with textual data. There are three main concepts to understand: analyzers, tokenizers, and filters.

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.

## Using Analyzers, Tokenizers and Filters

Although the analysis process is used for both indexing and querying, the same analysis process need not be used for both operations. 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.

## For More Information

These sections 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.

**For Analyzers, see:**

- Analyzers: Detailed conceptual information about Solr analyzers.
- Running Your Analyzer: Detailed information about testing and running your Solr analyzer.

**For Tokenizers, see:**

- About Tokenizers: Detailed conceptual information about Solr tokenizers.
- Tokenizers: Information about configuring tokenizers, and about the tokenizer factory classes included in this distribution of Solr.

**For Filters, see:**

- About Filters: Detailed conceptual information about Solr filters.
- Filter Descriptions: Information about configuring filters, and about the filter factory classes included in this distribution of Solr.
- CharFilterFactories: Information about filters for pre-processing input characters.

**To find out how to use Tokenizers and Filters with various languages, see:**

- Language Analysis: Information about tokenizers and filters for character set conversion or for use with specific languages.

## Analyzers

An analyzer examines the text of fields and generates a token stream. Analyzers are specified as a child of the `<fieldType>` element in the `schema.xml` configuration 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, the 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`.

### 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, and so on) 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.

## About Tokenizers

The job of a tokenizer is to break up a stream of text into tokens, where each token is (usually) a sub-sequence 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.

A `TypeTokenFilterFactory` is available that creates a `TypeTokenFilter` that filters tokens based on their TypeAttribute, which is set in `factory.getStopTypes`.

For a complete list of the available TokenFilters, see the section Tokenizers.

### When To use a CharFilter vs. a TokenFilter

There are several pairs of CharFilters and TokenFilters that have related (ie: `MappingCharFilter` and `ASCIIFoldingFilter`) or nearly identical (ie: `PatternReplaceCharFilterFactory` and `PatternReplaceFilterFactory`) functionality and it may not always be obvious which is the best choice.

The decision about which to use depends largely on which Tokenizer you are using, and whether you need to preprocess the stream of characters.

For example, suppose you have a tokenizer such as `StandardTokenizer` and although you are pretty happy with how it works overall, you want to customize how some specific characters behave. You could modify the rules and re-build your own tokenizer with JFlex, but it might be easier to simply map some of the characters before tokenization with a `CharFilter`.

## About Filters

Like tokenizers, filters consume input and produce a stream of tokens. 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, which breaks the field's text into tokens. Those tokens then pass through Solr's standard filter, which removes dots from acronyms, and performs a few other common operations. All the tokens are then set to lowercase, 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. Solr includes several language-specific stemmers created by the Snowball generator that are based on the Porter stemming algorithm. The generic Snowball Porter Stemmer Filter can be used to configure any of these language stemmers. 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.

## 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.

Tokenizers discussed in this section:

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 Solr.

For more information about Solr's tokenizers, see http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters.

### 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, including Internet domain names.
- The "@" character is among the set of token-splitting punctuation, so email addresses are **not** preserved as single tokens.

Note that words are split at hyphens.

The Standard Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: `<ALPHANUM>`, `<NUM>`, `<SOUTHEAST_ASIAN>`, `<IDEOGRAPHIC>`, and `<HIRAGANA>`.

**Factory class:** `solr.StandardTokenizerFactory`

**Arguments:**

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

**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"

### Classic Tokenizer

The Classic Tokenizer preserves the same behavior as the Standard Tokenizer of Solr versions 3.1 and previous. It does not use the Unicode standard annex UAX#29 word boundary rules that the Standard Tokenizer uses. 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.ClassicTokenizerFactory`

**Arguments:**

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

**Example:**

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

### Keyword Tokenizer

This tokenizer treats the entire text field as a single token.

**Factory class:** `solr.KeywordTokenizerFactory`

**Arguments:** None

**Example:**

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

### Letter Tokenizer

This tokenizer creates tokens from strings of contiguous letters, discarding all non-letter characters.

**Factory class:** `solr.LetterTokenizerFactory`

**Arguments:** None

**Example:**

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

**In:** "I can't."

**Out:** "I", "can", "t"

### 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"

### 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 >= `minGramSize`.

**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", "bicyc", "icyc", "icycl", "cycl", "cycle", "ycle"

### 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 is 1) The minimum n-gram size, must be > 0.

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

`side`: ("front" or "back", default is "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"

### ICU Tokenizer

This tokenizer processes multilingual text and tokenizes it appropriately based on its script attribute.

You can customize this tokenizer's behavior by specifying per-script rule files. To add per-script rules, add a `rulefiles` argument, which should contain a comma-separated list of `code:rulefile` pairs in the following format: four-letter ISO 15924 script code, followed by a colon, then a resource path. For example, to specify rules for Latin (script code "Latn") and Cyrillic (script code "Cyrl"), you would enter `Latn:my.Latin.rules.rbbi,Cyrl:my.Cyrillic.rules.rbbi`.

The default `solr.ICUTokenizerFactory` provides UAX#29 word break rules tokenization (like `solr.StandardTokenizer`), but also includes custom tailorings for Hebrew (specializing handling of double and single quotation marks), and for syllable tokenization for Khmer, Lao, and Myanmar.

**Factory class:** `solr.ICUTokenizerFactory`

**Arguments:**

`rulefile`: a comma-separated list of `code:rulefile` pairs in the following format: four-letter ISO 15924 script code, followed by a colon, then a resource path.

**Example:**

```
<analyzer>
    <!-- no customization -->
    <tokenizer class="solr.ICUTokenizerFactory"/>
  </analyzer>
```

```
<analyzer>
    <tokenizer class="solr.ICUTokenizerFactory"
    rulefiles="Latn:my.Latin.rules.rbbi,Cyrl:my.Cyrillic.rules.rbbi"
  />
  </analyzer>
```

### Path Hierarchy Tokenizer

This tokenizer creates synonyms from file path hierarchies.

**Factory class:** `solr.PathHierarchyTokenizerFactory`

**Arguments:**

`delimiter`: (character, no default) You can specify the file path delimiter and replace it with a delimiter you provide. This can be useful for working with backslash delimiters.

`replace`: (character, no default) Specifies the delimiter character Solr uses in the tokenized output.

**Example:**

```
<fieldType name="text_path" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.PathHierarchyTokenizerFactory" delimiter="\" replace="/"/>
  </analyzer>
</fieldType>
```

**In:** "c:\usr\local\apache"

**Out:** "c:", "c:/usr", "c:/usr/local", "c:/usr/local/apache"

### 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"

### UAX29 URL Email 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 top-level Internet domain names (validated against the white list in the IANA Root Zone Database when the tokenizer was generated); email addresses; `file : //`, `http(s)://`, and `ftp://` addr

esses; IPv4 and IPv6 addresses; and preserves them as a single token.

The UAX29 URL Email Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: `<ALPHANUM>`, `<NUM>`, `<URL>`, `<EMAIL>`, `<SOUTHEAST_ASIAN>`, `<IDEOGRAPHIC>`, and `<HIRAGANA>`.

**Factory class:** `solr.UAX29URLEmailTokenizerFactory`

**Arguments:**

`maxTokenLength`: (integer, default 255) Solr ignores tokens that exceed the number of characters specified by `maxTokenLength`.

**Example:**

```
<analyzer>
  <tokenizer class="solr.UAX29URLEmailTokenizerFactory"/>
</analyzer>
```

**In:** "Visit http://accarol.com/contact.htm?from=external&a=10 or e-mail bob.cratchet@accarol.com"

**Out:** "Visit", "http://accarol.com/contact.htm?from=external&a=10", "or", "email", "bob.cratchet@accarol.com"

### 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?"

### Related Topics

- TokenizerFactories

## Filter Descriptions

You configure each filter with a `<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.

Filters discussed in this section:

- ASCII Folding Filter
- Beider-Morse Filter
- Classic Filter
- Common Grams Filter
- Collation Key Filter

```
<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 Solr.

For more information about Solr's filters, see http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters.

### ASCII Folding Filter

This filter converts alphabetic, numeric, and symbolic Unicode characters which are not in the Basic Latin Unicode block (the first 127 ASCII characters) to their ASCII equivalents, if one exists. This filter converts characters from the following Unicode blocks:

- C1 Controls and Latin-1 Supplement (PDF)
- Latin Extended-A (PDF)
- Latin Extended-B (PDF)
- Latin Extended Additional (PDF)
- Latin Extended-C (PDF)
- Latin Extended-D (PDF)
- IPA Extensions (PDF)
- Phonetic Extensions (PDF)
- Phonetic Extensions Supplement (PDF)
- General Punctuation (PDF)
- Superscripts and Subscripts (PDF)
- Enclosed Alphanumerics (PDF)
- Dingbats (PDF)
- Supplemental Punctuation (PDF)
- Alphabetic Presentation Forms (PDF)
- Halfwidth and Fullwidth Forms (PDF)

**Factory class:** `solr.ASCIIFoldingFilterFactory`

**Arguments:** None

**Example:**

```
<analyzer>
  <filter class="solr.ASCIIFoldingFilterFactory"/>
</analyzer>
```

**In:** "á" (Unicode character 00E1)

**Out:** "a" (ASCII character 97)

### Beider-Morse Filter

Implements the Beider-Morse Phonetic Matching (BMPM) algorithm, which allows identification of similar names, even if they are spelled differently or in different languages. More information about how this works is available in the section on Phonetic Matching.

**Factory class:** `solr.BeiderMorseFilterFactory`

**Arguments:**

`nameType`: Types of names. Valid values are GENERIC, ASHKENAZI, or SEPHARDIC. If not processing Ashkenazi or Sephardic names, use GENERIC.

`ruleType`: Types of rules to apply. Valid values are APPROX or EXACT.

`concat`: Defines if multiple possible matches should be combined with a pipe ("|").

`languageSet`: The language set to use. The value "auto" will allow the Filter to identify the language, or a comma-separated list can be supplied.

**Example:**

```
<analyzer>
   <tokenizer class="solr.StandardTokenizerFactory"/>
   <filter class="solr.BeiderMorseFilterFactory" nameType="GENERIC" ruleType="APPROX"
      concat="true" languageSet="auto">
   </filter>
</analyzer>
```

### Classic Filter

This filter takes the output of the Classic Tokenizer and strips periods from acronyms and "'s" from possessives.

**Factory class:** `solr.ClassicFilterFactory`

**Arguments:** None

**Example:**

```
<analyzer>
  <tokenizer class="solr.ClassicTokenizerFactory"/>
  <filter class="solr.ClassicFilterFactory"/>
</analyzer>
```

**In:** "I.B.M. cat's can't"

**Tokenizer to Filter:** "I.B.M", "cat's", "can't"

**Out:** "IBM", "cat", "can't"

### Common Grams Filter

This filter creates word shingles by combining common tokens such as stop words with regular tokens. This is useful for creating phrase queries containing common words, such as "the cat." Solr normally ignores stop words in queried phrases, so searching for "the cat" would return all matches for the word "cat."

**Factory class:** `solr.CommonGramsFilterFactory`

**Arguments:**

`words`: (a common word file in .txt format) Provide the name of a common word file, such as `stopwords.txt`.

`format`: (optional) If the stopwords list has been formatted for Snowball, you can specify `format="snowball"` so Solr can read the stopwords file.

`ignoreCase`: (boolean) If true, the filter ignores the case of words when comparing them to the common word file. The default is false.

**Example:**

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

**In:** "the Cat"

**Tokenizer to Filter:** "the", "Cat"

**Out:** "the_cat"

### Collation Key Filter

Collation allows sorting of text in a language-sensitive way. It is usually used for sorting, but can also be used with advanced searches. We've covered this in much more detail in the section on Unicode Collation.

### 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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "four", "score", "and", "twenty"

**Out:** "four", "scor", "score", "twen", "twent", "twenty"

### English Minimal Stem Filter

This filter stems plural English words to their singular form.

**Factory class:** `solr.EnglishMinimalStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "dogs cats"

**Tokenizer to Filter:** "dogs", "cats"

**Out:** "dog", "cat"

### Hunspell Stem Filter

The Hunspell Stem Filter provides support for several languages. You must provide the dictionary (`.dic`) and rules (`.aff`) files for each language you wish to use with the Hunspell Stem Filter. You can download those language files here. Be aware that your results will vary widely based on the quality of the provided dictionary and rules files. For example, some languages have only a minimal word list with no morphological information. On the other hand, for languages that have no stemmer but do have an extensive dictionary file, the Hunspell stemmer may be a good choice.

**Factory class:** `solr.HunspellStemFilterFactory`

**Arguments:**

`dictionary`: (required) The path of a dictionary file.
`affix`: (required) The path of a rules file.
`ignoreCase`: (boolean) controls whether matching is case sensitive or not. The default is false.
`strictAffixParsing`: (boolean) controls whether the affix parsing is strict or not. If true, an error while reading an affix rule causes a ParseException, otherwise is ignored. The default is true.

**Example:**

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.HunspellStemFilterFactory"
    dictionary="en_GB.dic"
    affix="en_GB.aff"
    ignoreCase="true"
    strictAffixParsing="true" />
</analyzer>
```

**In:** "jump jumping jumped"

**Tokenizer to Filter:** "jump", "jumping", "jumped"

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

### 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"

**Tokenizer to Filter:** "A", "hyphen-", "ated", "word"

**Out:** "A", "hyphenated", "word"

### ICU Folding Filter

This filter is a custom Unicode normalization form that applies the foldings specified in Unicode Technical Report 30 in addition to the `NFKC_Casefold` normalization form as described in ICU Normalizer 2 Filter. This filter is a better substitute for the combined behavior of the ASCII Folding Filter, Lower Case Filter, and ICU Normalizer 2 Filter.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

**Factory class:** `solr.ICUFoldingFilterFactory`

**Arguments:** None

**Example:**

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

For detailed information on this normalization form, see http://www.unicode.org/reports/tr30/tr30-4.html.

### ICU Normalizer 2 Filter

This filter factory normalizes text according to one of five Unicode Normalization Forms as described in Unicode Standard Annex #15:

- NFC: (name="nfc" mode="compose") Normalization Form C, canonical decomposition
- NFD: (name="nfc" mode="decompose") Normalization Form D, canonical decomposition, followed by canonical composition
- NFKC: (name="nfkc" mode="compose") Normalization Form KC, compatibility decomposition
- NFKD: (name="nfkc" mode="decompose") Normalization Form KD, compatibility decomposition, followed by canonical composition
- NFKC_Casefold: (name="nfkc_cf" mode="compose") Normalization Form KC, with additional Unicode case folding. Using the ICU Normalizer 2 Filter is a better-performing substitution for the Lower Case Filter and NFKC normalization.

**Factory class:** `solr.ICUNormalizer2FilterFactory`

**Arguments:**

`name`: (string) The name of the normalization form; `nfc`, `nfd`, `nfkc`, `nfkd`, `nfkc_cf`

`mode`: (string) The mode of Unicode character composition and decomposition; `compose` or `decompose`

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ICUNormalizer2FilterFactory" name="nkc_cf" mode="compose"/>
</analyzer>
```

For detailed information about these Unicode Normalization Forms, see http://unicode.org/reports/tr15/.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

### ICU Transform Filter

This filter applies ICU Tranforms to text. This filter supports only ICU System Transforms. Custom rule sets are not supported.

**Factory class:** `solr.ICUTransformFilterFactory`

**Arguments:**

`id`: (string) The identifier for the ICU System Transform you wish to apply with this filter. For a full list of ICU System Transforms, see http://demo.icu-project.org/icu-bin/translit?TEMPLATE_FILE=data/translit_rule_main.html.

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ICUTransformFilterFactory" id="Traditional-Simplified"/>
</analyzer>
```

For detailed information about ICU Transforms, see http://userguide.icu-project.org/transforms/general.

To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

### 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. 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. The default is **false**.

**Example:**

Where `keepwords.txt` contains:

`happy`

`funny`

`silly`

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

**In:** "Happy, sad or funny"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "Happy", "sad", "or", "funny"

**Filter to Filter:** "happy", "sad", "or", "funny"

**Out:** "happy", "funny"

### KStem Filter

KStem is an alternative to the Porter Stem Filter for developers looking for a less aggressive stemmer. KStem was written by Bob Krovetz, ported to Lucene by Sergio Guzman-Lara (UMASS Amherst). This stemmer is only appropriate for English language text.

**Factory class:** `solr.KStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "jump jumping jumped"

**Tokenizer to Filter:** "jump", "jumping", "jumped"

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

### 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"

**Tokenizer to Filter:** "turn", "right", "at", "Albuquerque"

**Out:** "turn", "right"

### 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"

**Tokenizer to Filter:** "Down", "With", "CamelCase"

**Out:** "down", "with", "camelcase"

### Managed Stop Filter

This is specialized version of the  Stop Words Filter Factory that uses a set of stop words that are managed from a REST API.

**Arguments:**

`managed`: The name that should be used for this set of stop words in the managed REST API.

**Example:**

With this configuration the set of words is named "english" and can be managed via `/solr/[collection]/schema/analysis/stopwords/english`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedStopFilterFactory" managed="english"/>
</analyzer>
```

See Stop Filter for example input/output.

### Managed Synonym Filter

This is specialized version of the Synonym Filter Factory that uses a mapping on synonyms that is managed from a REST API.

**Arguments:**

`managed`: The name that should be used for this mapping on synonyms in the managed REST API.

**Example:**

With this configuration the set of mappings is named "english" and can be managed via `/solr/[collection]/schema/analysis/synonyms/english`

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ManagedSynonymFilterFactory" managed="english"/>
</analyzer>
```

See Synonym Filter for example input/output.

### N-Gram Filter

Generates n-gram tokens of sizes in the given range. Note that tokens are ordered by position and then by gram size.

**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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "four", "score"

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

### 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"

**Tokenizer to Filter:** "bing", "bang", "boom"

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

### 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.PatternReplaceFilterFactory`

**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.PatternReplaceFilterFactory" pattern="cat" replacement="dog"/>
</analyzer>
```

**In:** "cat concatenate catycat"

**Tokenizer to Filter:** "cat", "concatenate", "catycat"

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

**Example:**

String replacement, first occurrence only:

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

**In:** "cat concatenate catycat"

**Tokenizer to Filter:** "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.PatternReplaceFilterFactory" pattern="(\D+)(\d+)$"
replacement="$1_$2"/>
</analyzer>
```

**In:** "cat foo1234 9987 blah1234foo"

**Tokenizer to Filter:** "cat", "foo1234", "9987", "blah1234foo"

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

### 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", "RefinedSoundex", "Caverphone", or "ColognePhonetic"

`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.

`maxCodeLength`: (integer) The maximum length of the code to be generated by the Metaphone or Double Metaphone encoders.

**Example:**

Default behavior for DoubleMetaphone encoding.

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

**In:** "four score and twenty"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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)

### 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. But this stemmer is coded directly in Java and is not based on Snowball. It does not accept a list of protected words and is only appropriate for English language text. However, it has been benchmarked as four times faster than the English Snowball stemmer, so can provide a performance enhancement.

**Factory class:** `solr.PorterStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "jump jumping jumped"

**Tokenizer to Filter:** "jump", "jumping", "jumped"

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

### Position Filter Factory

This filter sets the position increment values of all tokens in a token stream except the first, which retains its original position increment value. This filter **has been deprecated** and will be removed in Solr 5.

**Factory class:** `solr.PositionIncrementFilterFactory`

**Arguments:**

`positionIncrement`: (integer, default = 0) The position increment value to apply to all tokens in a token stream except the first.

**Example:**

```
<analyzer>
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.PositionFilterFactory" positionIncrement="1"/>
</analyzer>
```

**In:** "hello world"

**Tokenizer to Filter:** "hello", "world"

**Out:** "hello" (token position 1), "world" (token position 2)

### 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:**

One example of where `RemoveDuplicatesTokenFilterFactory` is in situations where a synonym file is being used in conjuntion with a stemmer causes some synonyms to be reduced to the same stem. Consider the following entry from a `synonyms.txt` file:

```
Television, Televisions, TV, TVs
```

When used in the following configuration:

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

**In:** "Watch TV"

**Tokenizer to Synonym Filter:** "Watch"(1) "TV"(2)

**Synonym Filter to Stem Filter:** "Watch"(1) "Television"(2) "Televisions"(2) "TV"(2) "TVs"(2)

**Stem Filter to Remove Dups Filter:** "Watch"(1) "Television"(2) "Television"(2) "TV"(2) "TV"(2)

**Out:** "Watch"(1) "Television"(2) "TV"(2)

### Reversed Wildcard Filter

This filter reverses tokens to provide faster leading wildcard and prefix queries. Tokens without wildcards are not reversed.

**Factory class:** `solr.ReveresedWildcardFilterFactory`

**Arguments:**

`withOriginal` (boolean) If true, the filter produces both original and reversed tokens at the same positions. If false, produces only reversed tokens.

`maxPosAsterisk` (integer, default = 2) The maximum position of the asterisk wildcard ('*') that triggers the reversal of the query term. Terms with asterisks at positions above this value are not reversed.

`maxPosQuestion` (integer, default = 1) The maximum position of the question mark wildcard ('?') that triggers the reversal of query term. To reverse only pure suffix queries (queries with a single leading asterisk), set this to 0 and `maxPosAsterisk` to 1.

`maxFractionAsterisk` (float, default = 0.0) An additional parameter that triggers the reversal if asterisk ('*') position is less than this fraction of the query token length.

`minTrailing` (integer, default = 2) The minimum number of trailing characters in a query token after the last wildcard character. For good performance this should be set to a value larger than 1.

**Example:**

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.ReversedWildcardFilterFactory" withOriginal="true"
   maxPosAsterisk="2" maxPosQuestion="1" minTrailing="2" maxFractionAsterisk="0"/>
</analyzer>
```

**In:** "*foo *bar"

**Tokenizer to Filter:** "*foo", "*bar"

**Out:** "oof*", "rab*"

### 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:**

`minShingleSize`: (integer, default 2) The minimum number of tokens per shingle.

`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.

`outputUnigramsIfNoShingles`: (true/false) If false (the default), then individual tokens will be output if no shingles are possible.

`tokenSeparator`: (string, default is " ") The default string to use when joining adjacent tokens to form a shingle.

**Example:**

Default behavior.

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

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

**Tokenizer to Filter:** "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."

**Tokenizer to Filter:** "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)

### 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.

Solr contains Snowball stemmers for Armenian, Basque, Catalan, Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish and Turkish. For more information on Snowball, visit http://snowball.tartarus.org/.

`StopFilterFactory`, `CommonGramsFilterFactory`, and `CommonGramsQueryFilterFactory` can optionally read stopwords in Snowball format (specify `format="snowball"` in the configuration of those FilterFactories).

**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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "cante", "canta"

**Out:** "cant", "cant"

### 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

> ⚠ This filter is no longer operational in Solr when the `luceneMatchVersion` (in `solrconfig.xml`) is higher than "3.1".

### 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 to 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.

`format`: (optional) If the stopwords list has been formatted for Snowball, you can specify `format="snowball"` so Solr can read the stopwords file.

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

> ⚠ As of Solr 4.4, the `enablePositionIncrements` argument is no longer supported.

**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?"

**Tokenizer to Filter:** "To"(1), "be"(2), "or"(3), "what"(4)

**Out:** "To"(1), "what"(4)

**Example:**

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

**In:** "To be or what?"

**Tokenizer to Filter:** "To"(1), "be"(2), "or"(3), "what"(4)

**Out:** "what"(4)

### 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 is emitted in place of the token. The position value of the new tokens 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 a synonyms file named `mysynonyms.txt`:

```
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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "teh"(1), "ginormous"(2), "humungous"(3), "sofa"(4)

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

### 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"

**Tokenizer to Filter:** "bing", "bang", "boom"

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

### 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:** None

> ⚠ As of Solr 4.4, the `updateOffsets` argument is no longer supported.

**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 "

**Tokenizer to Filter:** "one", " two ", " three ", "four "

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

### 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."

**Tokenizer to Filter:** "Pay", "Bob's", "I.O.U."

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

### Type Token Filter

This filter blacklists or whitelists a specified list of token types, assuming the tokens have type metadata associated with them. For example, the UAX29 URL Email Tokenizer emits "<URL>" and "<EMAIL>" typed tokens, as well as other types. This filter would allow you to pull out only e-mail addresses from text as tokens, if you wish.

**Factory class:** `solr.TypeTokenFilterFactory`

**Arguments:**

`types`: Defines the location of a file of types to filter.

`useWhitelist`: If **true**, the file defined in `types` should be used as include list. If **false**, or undefined, the file defined in `types` is used as a blacklist.

> ⚠ As of Solr 4.4, the `enablePositionIncrements` argument is no longer supported.

**Example:**

```
<analyzer>
   <filter class="solr.TypeTokenFilterFactory" types="stoptypes.txt"
useWhitelist="true"/>
</analyzer>
```

### 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"`.
- 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" **->** "Bug Blaster", "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:"FemBot3000" **->** "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" **->** "1947 32"

catenateAll: (0/1, default 0) If non-zero, runs of word and number parts will be joined: "Zap-Master-9000" **->** "Zap Master9000"

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 through 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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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"

**Tokenizer to Filter:** "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-"

**Tokenizer to Filter:** "FooBar", "AstroBlaster", "XL-5000", "==ES-34-"

**Out:** "FooBar", "FooBar", "AstroBlaster", "XL-5000", "ES", "34"

### Related Topics

- TokenFilterFactories

## 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 while preserving the original character offsets to support features like highlighting.

Topics discussed in this section:

## 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.).

This filter requires specifying a `mapping` argument, which is the path and name of a file containing the mappings to perform.

Example:

```
<analyzer>
  <charFilter class="solr.MappingCharFilterFactory"
mapping="mapping-FoldToASCII.txt"/>
  <tokenizer ...>
  [...]
</analyzer>
```

### solr.HTMLStripCharFilterFactory

This filter creates `org.apache.solr.analysis.HTMLStripCharFilter`. This Char Filter strips HTML from the input stream and passes the result to another Char Filter or a 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 <!>.
- Removes contents of <script> and <style> elements.
- Handles XML comments inside these elements (normal comment processing will not always work).
- Replaces numeric character entities references like `&#65`; or `&#x7f`; with the corresponding character.
- The terminating ';' is optional if the entity reference at the end of the input; otherwise the terminating ';' is mandatory, to avoid false matches on something like "Alpha&Omega Corp".
- Replaces all named character entity references with the corresponding character.
- `&nbsp`; is replaced with a space instead of the 0xa0 character.
- Newlines are substituted for block-level elements.
- <CDATA> sections are recognized.
- Inline tags, such as `<b>`, `<i>`, or `<span>` will be removed.
- Uppercase character entities like `quot`, `gt`, `lt` and `amp` are recognized and handled as lowercase.

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 <a href="www.foo.bar">link</a>` | my link |
| `<br>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` | a<b A Alpha&Omega |

### solr.ICUNormalizer2CharFilterFactory

This filter performs pre-tokenization Unicode normalization using ICU4J.

Arguments:

`name`: A Unicode Normalization Form, one of `nfc`, `nfkc`, `nfkc_cf`. Default is `nfkc_cf`.

`mode`: Either `compose` or `decompose`. Default is `compose`. Use `decompose` with `name="nfc"` or `name="nfkc"` to get NFD or NFKD, respectively.

`filter`: A UnicodeSet pattern. Codepoints outside the set are always left unchanged. Default is `[]` (the null set, no filtering - all codepoints are subject to normalization).

Example:

```
<analyzer>
  <charFilter class="solr.ICUNormalizer2CharFilterFactory"/>
  <tokenizer ...>
  [...]
</analyzer>
```

### solr.PatternReplaceCharFilterFactory

This filter uses regular expressions to replace or change character patterns.

Arguments:

`pattern`: the regular expression pattern to apply to the incoming text.

`replacement`: the text to use to replace matching patterns.

You can configure this filter in `schema.xml` like this:

```
<analyzer>
  <charFilter class="solr.PatternReplaceCharFilterFactory"
             pattern="([nN][oO]\.)\s*(\d+)" replacement="$1$2"/>
  <tokenizer ...>
  [...]
</analyzer>
```

The table below presents examples of regex-based pattern replacement:

| Input | pattern | replacement | Output | Description |
|---|---|---|---|---|
| see-ing looking | `(\w+)(ing)` | $1 | see-ing look | Removes "ing" from the end of word. |
| see-ing looking | `(\w+)ing` | $1 | see-ing look | Same as above. 2nd parentheses can be omitted. |
| No.1 NO. no. 543 | `[nN][oO]\.\s*(\d+)` | #$1 | #1 NO. #543 | Replace some string literals |
| abc=1234=5678 | `(\w+)=(\d+)=(\d+)` | $3=$1=$2 | 5678=abc=1234 | Change the order of the groups. |

### Related Topics

- CharFilterFactories

## 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 white space 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.

For information about language detection at index time, see Detecting Languages During Indexing.

Topics discussed in this section:

### KeyWordMarkerFilterFactory

Protects words from being modified by stemmers. A customized protected word list may be specified with the "protected" attribute in the schema. Any words in the protected word list will not be modified by any stemmer in Solr.

A sample Solr `protwords.txt` with comments can be found in the `/solr/conf/` directory:

```
<fieldtype name="myfieldtype" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt" />
    <filter class="solr.PorterStemFilterFactory" />
  </analyzer>
</fieldtype>
```

### StemmerOverrideFilterFactory

Overrides stemming algorithms by applying a custom mapping, then protecting these terms from being modified by stemmers.

A customized mapping of words to stems, in a tab-separated file, can be specified to the "dictionary" attribute in the schema. Words in this mapping will be stemmed to the stems from the file, and will not be further changed by any stemmer.

A sample stemdict.txt with comments can be found in the Source Repository.

```
<fieldtype name="myfieldtype" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <filter class="solr.StemmerOverrideFilterFactory" dictionary="stemdict.txt" />
    <filter class="solr.PorterStemFilterFactory" />
  </analyzer>
</fieldtype>
```

### Dictionary Compound Word Token Filter

This filter splits, 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"

**Tokenizer to Filter:** "Donaudampfschiff"(1), "dummkopf"(2),

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

### Unicode Collation

Unicode Collation is a language-sensitive method of sorting text that can also be used for advanced search purposes.

Unicode Collation in Solr is fast, because all the work is done at index time.

Rather than specifying an analyzer within `<fieldtype ... class="solr.TextField">`, the `solr.CollationField` and `solr.ICUCollationField` field type classes provide this functionality. `solr.ICUCollationField`, which is backed by the ICU4J library, provides more flexible configuration, has more locales, is significantly faster, and requires less memory and less index space, since its keys are smaller than those produced by the JDK implementation that backs `solr.CollationField`.

`solr.ICUCollationField` is included in the Solr `analysis-extras` contrib - see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `SOLR_HOME/lib` in order to use it.

> ⚠ `CollationKeyFilterFactory` and `ICUCollationKeyFilterFactory` are deprecated token filter implementations of the same functionality as `solr.CollationField` and `solr.ICUCollationField`, respectively. These classes will no longer be available in Solr 5.0.

`solr.ICUCollationField` and `solr.CollationField` fields can be created in two ways:

- Based upon a system collator associated with a Locale.
- Based upon a tailored `RuleBasedCollator` ruleset.

**Arguments for `solr.ICUCollationField`, specified as attributes within the `<fieldtype>` element:**

Using a System collator:

`locale`: (required) RFC 3066 locale ID. See the ICU locale explorer for a list of supported locales.

`strength`: Valid values are `primary`, `secondary`, `tertiary`, `quaternary`, or `identical`. See Comparison Levels in ICU Collation Concepts for more information.

`decomposition`: Valid values are `no` or `canonical`. See Normalization in ICU Collation Concepts for more information.

Using a Tailored ruleset:

`custom`: (required) Path to a UTF-8 text file containing rules supported by the ICU `RuleBasedCollator`

`strength`: Valid values are `primary`, `secondary`, `tertiary`, `quaternary`, or `identical`. See Comparison

Levels in ICU Collation Concepts for more information.

`decomposition`: Valid values are `no` or `canonical`. See Normalization in ICU Collation Concepts for more information.

Expert options:

`alternate`: Valid values are `shifted` or `non-ignorable`. Can be used to ignore punctuation/whitespace.

`caseLevel`: (true/false) If true, in combination with `strength="primary"`, accents are ignored but case is taken into account. The default is false. See CaseLevel in ICU Collation Concepts for more information.

`caseFirst`: Valid values are `lower` or `upper`. Useful to control which is sorted first when case is not ignored.

`numeric`: (true/false) If true, digits are sorted according to numeric value, e.g. foobar-9 sorts before foobar-10. The default is false.

`variableTop`: Single character or contraction. Controls what is variable for `alternate`

### Sorting Text for a Specific Language

In this example, text is sorted according to the default German rules provided by ICU4J.

Locales are typically defined as a combination of language and country, but you can specify just the language if you want. For example, if you specify "de" as the language, you will get sorting that works well for the German language. If you specify "de" as the language and "CH" as the country, you will get German sorting specifically tailored for Switzerland.

```
<!-- Define a field type for German collation -->
<fieldType name="collatedGERMAN" class="solr.ICUCollationField"
           locale="de"
           strength="primary" />
...
<!-- Define a field to store the German collated manufacturer names. -->
<field name="manuGERMAN" type="collatedGERMAN" indexed="false" stored="false"
docValues="true"/>
...
<!-- Copy the text to this field. We could create French, English, Spanish versions
too,
     and sort differently for different users! -->
<copyField source="manu" dest="manuGERMAN"/>
```

In the example above, we defined the strength as "primary". The strength of the collation determines how strict the sort order will be, but it also depends upon the language. For example, in English, "primary" strength ignores differences in case and accents.

Another example:

```
<fieldType name="polishCaseInsensitive" class="solr.ICUCollationField"
           locale="pl_PL"
           strength="secondary" />
...
<field name="city" type="text_general" indexed="true" stored="true"/>
...
<field name="city_sort" type="polishCaseInsensitive" indexed="true" stored="false"/>
...
<copyField source="city" dest="city_sort"/>
```

The type will be used for the fields where the data contains Polish text. The "secondary" strength will ignore case differences, but, unlike "primary" strength, a letter with diacritic(s) will be sorted differently from the same base letter without diacritics.

An example using the "city_sort" field to sort:

```
q=*:*&fl=city&sort=city_sort+asc
```

### Sorting Text for Multiple Languages

There are two approaches to supporting multiple languages: if there is a small list of languages you wish to support, consider defining collated fields for each language and using `copyField`. However, adding a large number of sort fields can increase disk and indexing costs. An alternative approach is to use the Unicode `default` collator.

The Unicode `default` or `ROOT` locale has rules that are designed to work well for most languages. To use the `default` locale, simply define the locale as the empty string. This Unicode default sort is still significantly more advanced than the standard Solr sort.

```
<fieldType name="collatedROOT" class="solr.ICUCollationField"
           locale=""
           strength="primary" />
```

### Sorting Text with Custom Rules

You can define your own set of sorting rules. It's easiest to take existing rules that are close to what you want and customize them.

In the example below, we create a custom rule set for German called DIN 5007-2. This rule set treats umlauts in German differently: it treats ö as equivalent to oe, ä as equivalent to ae, and ü as equivalent to ue. For more information, see the ICU RuleBasedCollator javadocs.

This example shows how to create a custom rule set for `solr.ICUCollationField` and dump it to a file:

```
// get the default rules for Germany
// these are called DIN 5007-1 sorting
RuleBasedCollator baseCollator = (RuleBasedCollator) Collator.getInstance(new
ULocale("de", "DE"));

// define some tailorings, to make it DIN 5007-2 sorting.
// For example, this makes ö equivalent to oe
String DIN5007_2_tailorings =
    "& ae , a\u0308 & AE , A\u0308"+
    "& oe , o\u0308 & OE , O\u0308"+
    "& ue , u\u0308 & UE , u\u0308";

// concatenate the default rules to the tailorings, and dump it to a String
RuleBasedCollator tailoredCollator = new RuleBasedCollator(baseCollator.getRules() +
DIN5007_2_tailorings);
String tailoredRules = tailoredCollator.getRules();

// write these to a file, be sure to use UTF-8 encoding!!!
FileOutputStream os = new FileOutputStream(new
File("/solr_home/conf/customRules.dat"));
IOUtils.write(tailoredRules, os, "UTF-8");
```

This rule set can now be used for custom collation in Solr:

```
<fieldType name="collatedCUSTOM" class="solr.ICUCollationField"
           custom="customRules.dat"
           strength="primary" />
```

#### JDK Collation

As mentioned above, ICU Unicode Collation is better in several ways than JDK Collation, but if you cannot use ICU4J for some reason, you can use `solr.CollationField`.

The principles of JDK Collation are the same as those of ICU Collation; you just specify `language`, `country` and `variant` arguments instead of the combined `locale` argument.

**Arguments for `solr.CollationField`, specified as attributes within the `<fieldtype>` element:**

Using a System collator (see Oracle's list of locales supported in Java 7):

`language`: (required) ISO-639 language code

`country`: ISO-3166 country code

`variant`: Vendor or browser-specific code

`strength`: Valid values are `primary`, `secondary`, `tertiary` or `identical`. See Oracle Java 7 Collator javadocs for more information.

`decomposition`: Valid values are `no`, `canonical`, or `full`. See Oracle Java 7 Collator javadocs for more information.

Using a Tailored ruleset:

`custom`: (required) Path to a UTF-8 text file containing rules supported by the `JDK RuleBasedCollator` `strength`: Valid values are `primary`, `secondary`, `tertiary` or `identical`. See Oracle Java 7 Collator javadocs for more information.

`decomposition`: Valid values are `no`, `canonical`, or `full`. See Oracle Java 7 Collator javadocs for more information.

**A `solr.CollationField` example:**

```
<fieldType name="collatedGERMAN" class="solr.CollationField"
           language="de"
           country="DE"
           strength="primary" /> <!-- ignore Umlauts and letter case when sorting -->
...
<field name="manuGERMAN" type="collatedGERMAN" indexed="false" stored="false"
docValues="true" />
...
<copyField source="manu" dest="manuGERMAN"/>
```

### ASCII Folding Filter

This filter converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists. Only those characters with reasonable ASCII alternatives are converted:

This can increase recall by causing more matches. On the other hand, it can reduce precision because language-specific character differences may be lost.

**Factory class:** `solr.ASCIIFoldingFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "Björn Ångström"

**Tokenizer to Filter:** "Björn", "Ångström"

**Out:** "Bjorn", "Angstrom"

### Language-Specific Factories

These factories are each designed to work with specific languages. The languages covered here are:

- Arabic
- Brazilian Portuguese
- Bulgarian
- Catalan
- Chinese
- Simplified Chinese
- Danish
- Dutch
- Finnish
- French
- Galician
- German
- Greek
- Hebrew,
- Hindi
- Indonesian
- Italian
- Irish
- Kuromoji (Japanese)
- Latvian
- Norwegian
- Polish
- Portuguese
- Romanian
- Russian
- Spanish
- Swedish
- Thai
- Turkish

- CJK
- Czech

Lao, Myanmar, Khmer

- Persian

### Arabic

Solr provides support for the Light-10 (PDF) stemming algorithm, and Lucene includes an example stopword list.

This algorithm defines both character normalization and stemming, so these are split into two filters to provide more flexibility.

**Factory classes:** `solr.ArabicStemFilterFactory`, `solr.ArabicNormalizationFilterFactory`

**Arguments:** None

**Example:**

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

### Brazilian Portuguese

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"

**Tokenizer to Filter:** "praia", "praias"

**Out:** "pra", "pra"

### Bulgarian

Solr includes a light stemmer for Bulgarian, following this algorithm (PDF), and Lucene includes an example stopword list.

**Factory class:** `solr.BulgarianStemFilterFactory`

**Arguments:** None

**Example:**

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

### Catalan

Solr can stem Catalan using the Snowball Porter Stemmer with an argument of `language="Catalan"`. Solr includes a set of contractions for Catalan, which can be stripped using `solr.ElisionFilterFactory`.

**Factory class:** `solr.SnowballPorterFilterFactory`

**Arguments:**

`language`: (required) stemmer language, "Catalan" in this case

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_ca.txt"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Catalan" />
</analyzer>
```

**In:** "llengües llengua"

**Tokenizer to Filter:** "llengües"(1) "llengua"(2),

**Out:** "llengu"(1), "llengu"(2)

### Chinese

#### *Chinese Tokenizer*

The Chinese Tokenizer is deprecated as of Solr 3.4. Use the `solr.StandardTokenizerFactory` instead.

**Factory class:** `solr.ChineseTokenizerFactory`

**Arguments:** None

**Example:**

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

#### *Chinese Filter Factory*

The Chinese Filter Factory is deprecated as of Solr 3.4. Use the `solr.StopFilterFactory` instead.

**Factory class:** `solr.ChineseFilterFactory`

**Arguments:** None

**Example:**

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

#### Simplified Chinese

For Simplified Chinese, Solr provides support for Chinese sentence and word segmentation with the `solr.HMMChineseTokenizerFactory` in the `analysis-extras` contrib module. This component includes a large dictionary and segments Chinese text into words with the Hidden Markov Model. To use this filter, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

**Factory class:** `solr.HMMChineseTokenizerFactory`

**Arguments:** None

**Examples:**

To use the default setup with fallback to English Porter stemmer for English words, use:

`<analyzer class="org.apache.lucene.analysis.cn.smart.SmartChineseAnalyzer"/>`

Or to configure your own analysis setup, use the `solr.HMMChineseTokenizerFactory` along with your custom filter setup.

```
<analyzer>
  <tokenizer class="solr.HMMChineseTokenizerFactory"/>
  <filter class="solr.StopFilterFactory
          words="org/apache/lucene/analysis/cn/smart/stopwords.txt"/>
  <filter class="solr.PorterStemFilterFactory"/>
</analyzer>
```

#### CJK

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>
```

#### Czech

Solr includes a light stemmer for Czech, following this algorithm, and Lucene includes an example stopword list.

**Factory class:** `solr.CzechStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "prezidenští, prezidenta, prezidentského"

**Tokenizer to Filter:** "prezidenští", "prezidenta", "prezidentského"

**Out:** "preziden", "preziden", "preziden"

### Danish

Solr can stem Danish using the Snowball Porter Stemmer with an argument of `language="Danish"`.

**Factory class:** `solr.SnowballPorterFilterFactory`

**Arguments:**

`language`: (required) stemmer language, "Danish" in this case

**Example:**

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

**In:** "undersøg undersøgelse"

**Tokenizer to Filter:** "undersøg"(1) "undersøgelse"(2),

**Out:** "undersøg"(1), "undersøg"(2)

### Dutch

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"`.

**Factory class:** `solr.DutchStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "kanaal kanalen"

**Tokenizer to Filter:** "kanaal", "kanalen"

**Out:** "kanal", "kanal"

### Finnish

Solr includes support for stemming Finnish, and Lucene includes an example stopword list.

**Factory class:** `solr.FinnishLightStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "kala kalat"

**Tokenizer to Filter:** "kala", "kalat"

**Out:** "kala", "kala"

### French

#### *Elision Filter*

Removes article elisions from a token stream. This filter can be useful for languages such as French, Catalan, Italian, and Irish.

**Factory class:** `solr.ElisionFilterFactory`

**Arguments:**

`articles`: 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 in *l'avion* (the plane). This file should include the abbreviated form, which precedes the apostrophe. In this case, simply "*l*". If no `articles` attribute is specified, a default set of French articles is used.

`ignoreCase`: (boolean) If true, the filter ignores the case of words when comparing them to the common word file. Defaults to `false`

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ElisionFilterFactory"
          ignoreCase="true"
          articles="lang/contractions_fr.txt"/>
</analyzer>
```

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

**Tokenizer to Filter:** "L'histoire", "d'art"

**Out:** "histoire", "art"

#### *French Light Stem Filter*

Solr includes three stemmers for French: one in the `solr.SnowballPorterFilterFactory`, a lighter stemmer called `solr.FrenchLightStemFilterFactory`, and an even less aggressive stemmer called `solr.FrenchMinimalStemFilterFactory`. Lucene includes an example stopword list.

**Factory classes:** `solr.FrenchLightStemFilterFactory`, `solr.FrenchMinimalStemFilterFactory`

**Arguments:** None

**Examples:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_fr.txt"/>
  <filter class="solr.FrenchLightStemFilterFactory"/>
</analyzer>
```

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_fr.txt"/>
  <filter class="solr.FrenchMinimalStemFilterFactory"/>
</analyzer>
```

**In:** "le chat, les chats"

**Tokenizer to Filter:** "le", "chat", "les", "chats"

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

### Galician

Solr includes a stemmer for Galician following this algorithm, and Lucene includes an example stopword list.

**Factory class:** `solr.GalicianStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "felizmente Luzes"

**Tokenizer to Filter:** "felizmente", "luzes"

**Out:** "feliz", "luz"

### German

Solr includes four stemmers for German: one in the `solr.SnowballPorterFilterFactory language="German"`, a stemmer called `solr.GermanStemFilterFactory`, a lighter stemmer called `solr.GermanLightStemFilterFactory`, and an even less aggressive stemmer called `solr.GermanMinimalStemFilterFactory`. Lucene includes an example stopword list.

**Factory classes:** `solr.GermanStemFilterFactory`, `solr.LightGermanStemFilterFactory`, `solr.MinimalGermanStemFilterFactory`

**Arguments:** None

**Examples:**

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

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

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

**In:** "hund hunden"

**Tokenizer to Filter:** "hund", "hunden"

**Out:** "hund", "hund"

### Greek

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

**Factory class:** `solr.GreekLowerCaseFilterFactory`

**Arguments:** None

> ⚠ Use of custom charsets is not longer supported as of Solr 3.1. If you need to index text in these encodings, please use Java's character set conversion facilities (InputStreamReader, and so on.) during I/O, so that Lucene can analyze this text as Unicode instead.

**Example:**

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

### Hindi

Solr includes support for stemming Hindi following this algorithm (PDF), support for common spelling differences through the `solr.HindiNormalizationFilterFactory`, support for encoding differences through the `solr.IndicNormalizationFilterFactory` following this algorithm, and Lucene includes an example stopword list.

**Factory classes:** `solr.IndicNormalizationFilterFactory`, `solr.HindiNormalizationFilterFactory`, `solr.HindiStemFilterFactory`

**Arguments:** None

**Example:**

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.IndicNormalizationFilterFactory"/>
  <filter class="solr.HindiNormalizationFilterFactory"/>
  <filter class="solr.HindiStemFilterFactory"/>
</analyzer>
```

### Indonesian

Solr includes support for stemming Indonesian (Bahasa Indonesia) following this algorithm (PDF), and Lucene includes an example stopword list.

**Factory class:** `solr.IndonesianStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "sebagai sebagainya"

**Tokenizer to Filter:** "sebagai", "sebagainya"

**Out:** "bagai", "bagai"

#### Italian

Solr includes two stemmers for Italian: one in the `solr.SnowballPorterFilterFactory language="Italian"`, and a lighter stemmer called `solr.ItalianLightStemFilterFactory`. Lucene includes an example stopword list.

**Factory class:** `solr.ItalianStemFilterFactory`

**Arguments:** None

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_it.txt"/>
  <filter class="solr.ItalianLightStemFilterFactory"/>
</analyzer>
```

**In:** "propaga propagare propagamento"

**Tokenizer to Filter:** "propaga", "propagare", "propagamento"

**Out:** "propag", "propag", "propag"

#### Irish

Solr can stem Irish using the Snowball Porter Stemmer with an argument of `language="Irish"`. Solr includes `solr.IrishLowerCaseFilter`, which can handle Irish-specific constructs. Solr also includes a set of contractions for Irish which can be stripped using `solr.ElisionFilterFactory`.

**Factory class:** `solr.SnowballPorterFilterFactory`

**Arguments:**

`language`: (required) stemmer language, "Irish" in this case

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ElisionFilterFactory"
          articles="lang/contractions_ga.txt"/>
  <filter class="solr.IrishLowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Irish" />
</analyzer>
```

**In:** "siopadóireacht síceapatacha b'fhearr m'athair"

**Tokenizer to Filter:** "siopadóireacht", "síceapatacha", "b'fhearr", "m'athair"

**Out:** "siopadóir", "síceapaite", "fearr", "athair"

#### Kuromoji (Japanese)

Solr includes support for stemming Kuromoji (Japanese), and Lucene includes an example stopword list. Kuromoji has a search mode (default) that does segmentation useful for search. A heuristic is used to segment compounds into its parts and the compound itself is kept as a synonym.

With Solr 4, the `JapaneseIterationMarkCharFilterFactory` now is included to normalize Japanese iteration marks.

You can also make discarding punctuation configurable in the `JapaneseTokenizerFactory`, by setting `discardPunctuation` to `false` (to show punctuation) or `true` (to discard punctuation).

**Factory class:** `solr.KuromojiStemFilterFactory`

**Arguments:**

`mode`: Use search-mode to get a noun-decompounding effect useful for search. Search mode improves segmentation for search at the expense of part-of-speech accuracy. Valid values for mode are:

- `normal`: default segmentation
- `search`: segmentation useful for search (extra compound splitting)
- `extended`: search mode with unigramming of unknown words (experimental)

For some applications it might be good to use search mode for indexing and normal mode for queries to reduce recall and prevent parts of compounds from being matched and highlighted.

Kuromoji also has a convenient user dictionary feature that allows overriding the statistical model with your own entries for segmentation, part-of-speech tags and readings without a need to specify weights. Note that user dictionaries have not been subject to extensive testing. User dictionary attributes are:

`userDictionary`: user dictionary filename
`userDictionaryEncoding`: user dictionary encoding (default is UTF-8)

See `lang/userdict_ja.txt` for a sample user dictionary file.

Punctuation characters are discarded by default. Use `discardPunctuation="false"` to keep them.

**Example:**

```
<fieldType name="text_ja" positionIncrementGap="100"
autoGeneratePhraseQueries="false">
   <analyzer>
      <tokenizer class="solr.JapaneseTokenizerFactory" mode="search"
userDictionary="lang/userdict_ja.txt"/>
      <filter class="solr.JapaneseBaseFormFilterFactory"/>
      <filter class="solr.JapanesePartOfSpeechStopFilterFactory"
tags="lang/stoptags_ja.txt" enablePositionIncrements="true"/>
      <filter class="solr.CJKWidthFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_ja.txt" enablePositionIncrements="true" />
      <filter class="solr.JapaneseKatakanaStemFilterFactory" minimumLength="4"/>
      <filter class="solr.LowerCaseFilterFactory"/>
   </analyzer>
</fieldType>
```

### Hebrew, Lao, Myanmar, Khmer

Lucene provides support, in addition to UAX#29 word break rules, for Hebrew's use of the double and single quote characters, and for segmenting Lao, Myanmar, and Khmer into syllables with the `solr.ICUTokenizerFactory` in the analysis-extras contrib module. To use this tokenizer, see solr/contrib/analysis-extras/README.txt for instructions on which jars you need to add to your solr_home/lib.

See the ICUTokenizer for more information.

### Latvian

Solr includes support for stemming Latvian, and Lucene includes an example stopword list.

**Factory class:** solr.LatvianStemFilterFactory

**Arguments:** None

**Example:**

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

**In:** "tirgiem tirgus"

**Tokenizer to Filter:** "tirgiem", "tirgus"

**Out:** "tirg", "tirg"

### Norwegian

Solr includes two classes for stemming Norwegian, NorwegianLightStemFilterFactory and NorwegianMinimalStemFilterFactory. Lucene includes an example stopword list.

Another option is to use the Snowball Porter Stemmer with an argument of language="Norwegian".

#### *Norwegian Light Stemmer*

The NorwegianLightStemFilterFactory requires a "two-pass" sort for the -dom and -het endings. This means that in the first pass the word "kristendom" is stemmed to "kristen", and then all the general rules apply so it will be further stemmed to "krist". The effect of this is that "kristen," "kristendom," "kristendommen," and "kristendommens" will all be stemmed to "krist."

The second pass is to pick up -dom and -het endings. Consider this example:

| **One pass** | | **Two passes** | |
|---|---|---|---|
| **Before** | **After** | **Before** | **After** |
| forlegen | forleg | forlegen | forleg |
| forlegenhet | forlegen | forlegenhet | forleg |
| forlegenheten | forlegen | forlegenheten | forleg |
| forlegenhetens | forlegen | forlegenhetens | forleg |

| | | | |
|---|---|---|---|
| firkantet | firkant | firkantet | firkant |
| firkantethet | firkantet | firkantethet | firkant |
| firkantetheten | firkantet | firkantetheten | firkant |

**Factory class:** `solr.NorwegianLightStemFilterFactory`

**Arguments:** `variant:` Choose the Norwegian language variant to use. Valid values are:

- `nb:` Bokmål (default)
- `nn:` Nynorsk
- `no:` both

### Example:

```
<fieldType name="text_no" class="solr.TextField" positionIncrementGap="100">
   <analyzer>
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_no.txt" format="snowball" enablePositionIncrements="true"/>
      <filter class="solr.NorwegianLightStemFilterFactory"/>
   </analyzer>
</fieldType>
```

**In:** "Forelskelsen"

**Tokenizer to Filter:** "forelskelsen"

**Out:** "forelske"

#### *Norwegian Minimal Stemmer*

The `NorwegianMinimalStemFilterFactory` stems plural forms of Norwegian nouns only.

**Factory class:** `solr.NorwegianMinimalStemFilterFactory`

**Arguments:** `variant:` Choose the Norwegian language variant to use. Valid values are:

- `nb:` Bokmål (default)
- `nn:` Nynorsk
- `no:` both

### Example:

```
<fieldType name="text_no" class="solr.TextField" positionIncrementGap="100">
   <analyzer>
      <tokenizer class="solr.StandardTokenizerFactory"/>
      <filter class="solr.LowerCaseFilterFactory"/>
      <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_no.txt" format="snowball" enablePositionIncrements="true"/>
      <filter class="solr.NorwegianMinimalStemFilterFactory"/>
   </analyzer>
</fieldType>
```

**In:** "Bilens"

**Tokenizer to Filter:** "bilens"

**Out:** "bil"

### Persian

#### *Persian Filter Factories*

Solr includes support for normalizing Persian, and Lucene includes an example stopword list.

**Factory class:** `solr.PersianNormalizationFilterFactory`

**Arguments:** None

**Example:**

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

### Polish

Solr provides support for Polish stemming with the `solr.StempelPolishStemFilterFactory`, and `solr.MorphologikFilterFactory` for lemmatization, in the `contrib/analysis-extras` module. The `solr.StempelPolishStemFilterFactory` component includes an algorithmic stemmer with tables for Polish. To use either of these filters, see `solr/contrib/analysis-extras/README.txt` for instructions on which jars you need to add to your `solr_home/lib`.

**Factory class:** `solr.StempelPolishStemFilterFactory` and `solr.MorfologikFilterFactory`

**Arguments:** None

**Example:**

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

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.MorfologikFilterFactory" dictionary-resource="pl"/>
</analyzer>
```

**In:** ""studenta studenci"

**Tokenizer to Filter:** "studenta", "studenci"

**Out:** "student", "student"

More information about the Stempel stemmer is available in the Lucene javadocs.

The Morfologik `dictionary-resource` param value is a constant specifying which dictionary to choose. The dictionary resource must be named `morfologik/dictionaries/{dictionaryResource}.dict` and have an associated `.info` metadata file. See the Morfologik project for details.

### Portuguese

Solr includes four stemmers for Portuguese: one in the `solr.SnowballPorterFilterFactory`, an alternative stemmer called `solr.PortugueseStemFilterFactory`, a lighter stemmer called `solr.PortugueseLightStemFilterFactory`, and an even less aggressive stemmer called `solr.PortugueseMinimalStemFilterFactory`. Lucene includes an example stopword list.

**Factory classes:** `solr.PortugueseStemFilterFactory`, `solr.PortugueseLightStemFilterFactory`, `solr.PortugueseMinimalStemFilterFactory`

**Arguments:** None

**Example:**

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

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

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

**In:** "praia praias"

**Tokenizer to Filter:** "praia", "praias"

**Out:** "pra", "pra"

### Romanian

Solr can stem Romanian using the Snowball Porter Stemmer with an argument of `language="Romanian"`.

**Factory class:** `solr.SnowballPorterFilterFactory`

**Arguments:**

`language`: (required) stemmer language, "Romanian" in this case

**Example:**

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

### Russian

#### *Russian Stem Filter*

Solr includes two stemmers for Russian: one in the `solr.SnowballPorterFilterFactory language="Russian"`, and a lighter stemmer called `solr.RussianLightStemFilterFactory`. Lucene includes an example stopword list.

**Factory class:** `solr.RussianLightStemFilterFactory`

**Arguments:** None

> Use of custom charsets is no longer supported as of Solr 3.4. If you need to index text in these encodings, please use Java's character set conversion facilities (InputStreamReader, and so on.) during I/O, so that Lucene can analyze this text as Unicode instead.

**Example:**

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

### Spanish

Solr includes two stemmers for Spanish: one in the `solr.SnowballPorterFilterFactory language="Spanish"`, and a lighter stemmer called `solr.SpanishLightStemFilterFactory`. Lucene includes an example stopword list.

**Factory class:** `solr.SpanishStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "torear toreara torearlo"

**Tokenizer to Filter:** "torear", "toreara", "torearlo"

**Out:** "tor", "tor", "tor"

## Swedish

### *Swedish Stem Filter*

Solr includes two stemmers for Swedish: one in the `solr.SnowballPorterFilterFactory language="Swedish"`, and a lighter stemmer called `solr.SwedishLightStemFilterFactory`. Lucene includes an example stopword list.

**Factory class:** `solr.SwedishStemFilterFactory`

**Arguments:** None

**Example:**

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

**In:** "kloke klokhet klokheten"

**Tokenizer to Filter:** "kloke", "klokhet", "klokheten"

**Out:** "klok", "klok", "klok"

## Thai

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.ThaiTokenizerFactory`

**Arguments:** None

**Example:**

```
<analyzer type="index">
  <tokenizer class="solr.ThaiTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
</analyzer>
```

## Turkish

Solr includes support for stemming Turkish through the `solr.SnowballPorterFilterFactory`; support for case-insensitive search through the `solr.TurkishLowerCaseFilterFactory`; support for stripping apostrophes and following suffixes through `solr.ApostropheFilterFactory` (see Role of Apostrophes in Turkish Information Retrieval); support for a form of stemming that truncating tokens at a configurable maximum length through the solr.TruncateTokenFilterFactory (see Information Retrieval on Turkish Texts); and Lucene includes an example stopword list.

**Factory class:** `solr.TurkishLowerCaseFilterFactory`

**Arguments:** None

**Example:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ApostropheFilterFactory"/>
  <filter class="solr.TurkishLowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Turkish"/>
</analyzer>
```

**Another example, illustrating diacritics-insensitive search:**

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.ApostropheFilterFactory"/>
  <filter class="solr.TurkishLowerCaseFilterFactory"/>
  <filter class="solr.ASCIIFoldingFilterFactory" preserveOriginal="true"/>
  <filter class="solr.KeywordRepeatFilterFactory"/>
  <filter class="solr.TruncateTokenFilterFactory" prefixLength="5"/>
  <filter class="solr.RemoveDuplicatesTokenFilterFactory"/>
</analyzer>
```

### Related Topics

- LanguageAnalysis

## Phonetic Matching

Introduced with Solr v3.6, Beider-Morse Phonetic Matching (BMPM) is a "soundalike" tool that lets you search using a new phonetic matching system. BMPM helps you search for personal names (or just surnames) in a Solr/Lucene index, and is far superior to the existing phonetic codecs, such as regular soundex, metaphone, caverphone, etc.

In general, phonetic matching lets you search a name list for names that are phonetically equivalent to the desired name. BMPM is similar to a soundex search in that an exact spelling is not required. Unlike soundex, it does not generate a large quantity of false hits.

From the spelling of the name, BMPM attempts to determine the language. It then applies phonetic rules for that particular language to transliterate the name into a phonetic alphabet. If it is not possible to determine the language with a fair degree of certainty, it uses generic phonetic instead. Finally, it applies language-independent rules regarding such things as voiced and unvoiced consonants and vowels to further insure the reliability of the matches.

For example, assume that the matches found when searching for Stephen in a database are "Stefan", "Steph", "Stephen", "Steve", "Steven", "Stove", and "Stuffin". "Stefan", "Stephen", and "Steven" are probably relevant, and are names that you want to see. "Stuffin", however, is probably not relevant. Also rejected were "Steph", "Steve", and "Stove". Of those, "Stove" is probably not one that we would have wanted. But "Steph" and "Steve" are possibly ones that you might be interested in.

For Solr, BMPM searching is available for the following languages:

- English
- French
- German
- Greek
- Hebrew written in Hebrew letters
- Hungarian
- Italian
- Lithuanian and Latvian
- Polish
- Romanian
- Russian written in Cyrillic letters
- Russian transliterated into English letters
- Spanish
- Turkish

The name matching is also applicable to non-Jewish surnames from the countries in which those languages are spoken.

For more information, see here: http://stevemorse.org/phoneticinfo.htm and http://stevemorse.org/phonetics/bmpm.htm.

## Running Your Analyzer

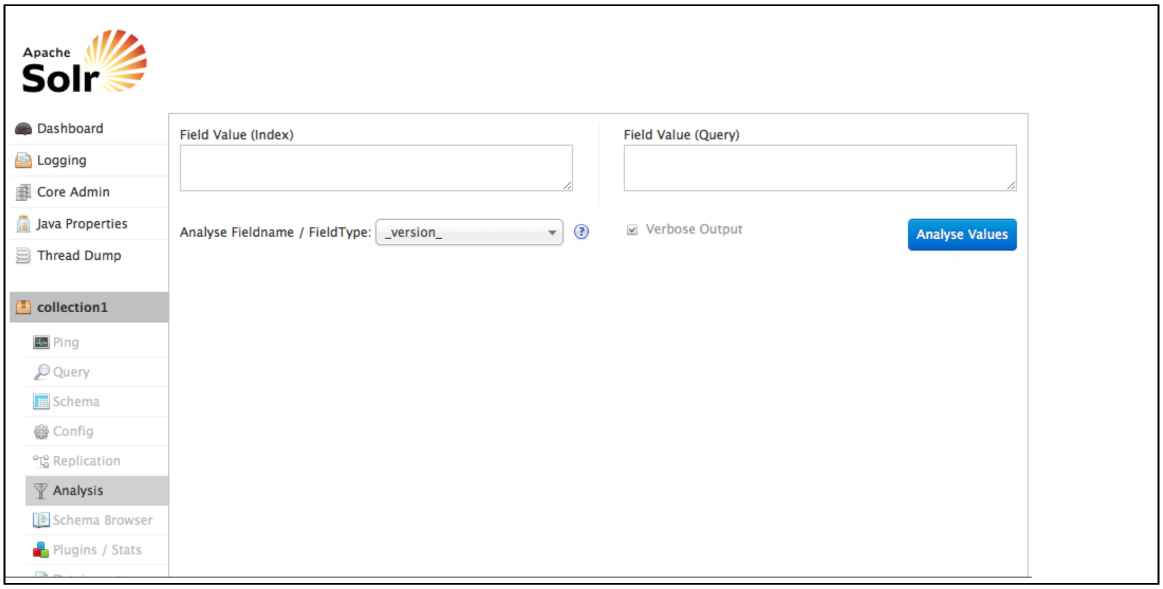
Once you've defined a field type in `schema.xml` and specified the analysis steps that you want applied to it, you should test it out to make sure that it behaves the way you expect it to. Luckily, there is a very handy page in the Solr admin interface that lets you do just that. You can invoke the analyzer 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, which may have been split across lines in the text, then to set all words to lowercase. For queries, you want to skip the de-hyphenation step.

To test this out, point your browser at the Analysis Screen 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/#/collection1/analysis. You should see a page like this.



Analyse Fieldname / FieldType: _version_

☑ Verbose Output

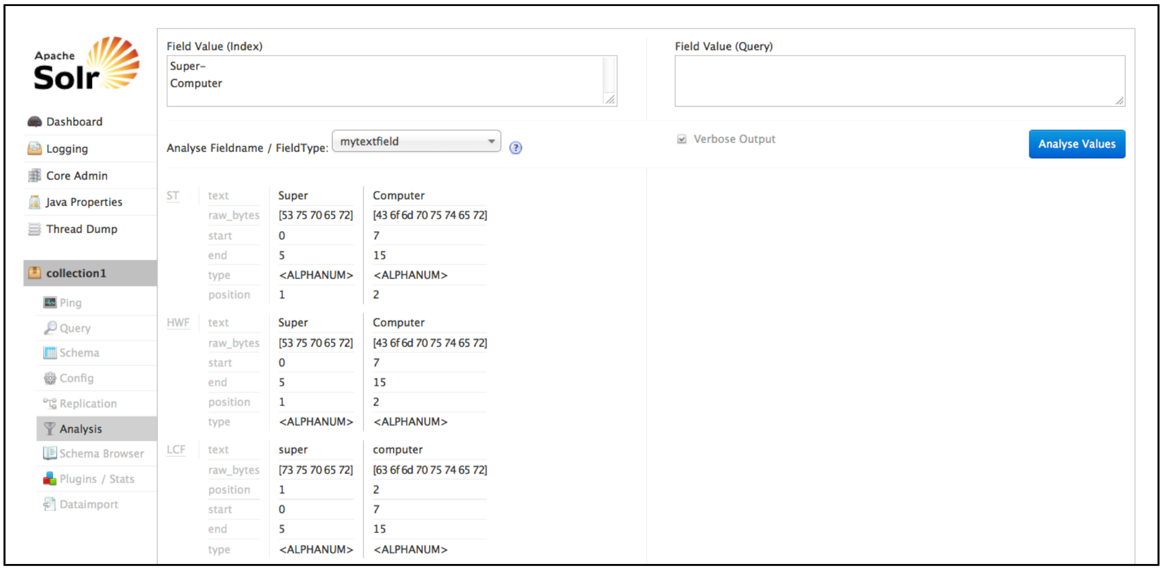
*Empty Analysis screen*

We want to test the field type definition for "mytextfield", defined above. The drop-down labeled "Analyse

Fieldname/FieldType" allows choosing the field or field type to use for the analysis.

There are two "Field Value" boxes, one for how text will be analyzed during indexing and a second for how text will be analyzed for query processing. In the "Field Value (Index)" box enter some sample text "Super-computer" in this example) to be processed by the analyzer. 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:



Field Value (Index)

Super-
Computer

Field Value (Query)

Analyse Fieldname / FieldType: mytextfield

☑ Verbose Output

|  |  |  |  |
|---|---|---|---|
| ST | text | Super | Computer |
|  | raw_bytes | [53 75 70 65 72] | [43 6f 6d 70 75 74 65 72] |
|  | start | 0 | 7 |
|  | end | 5 | 15 |
|  | type | <ALPHANUM> | <ALPHANUM> |
|  | position | 1 | 2 |
| HWF | text | Super | Computer |
|  | raw_bytes | [53 75 70 65 72] | [43 6f 6d 70 75 74 65 72] |
|  | start | 0 | 7 |
|  | end | 5 | 15 |
|  | position | 1 | 2 |
|  | type | <ALPHANUM> | <ALPHANUM> |
| LCF | text | super | computer |
|  | raw_bytes | [73 75 70 65 72] | [63 6f 6d 70 75 74 65 72] |
|  | position | 1 | 2 |
|  | start | 0 | 7 |
|  | end | 5 | 15 |
|  | type | <ALPHANUM> | <ALPHANUM> |

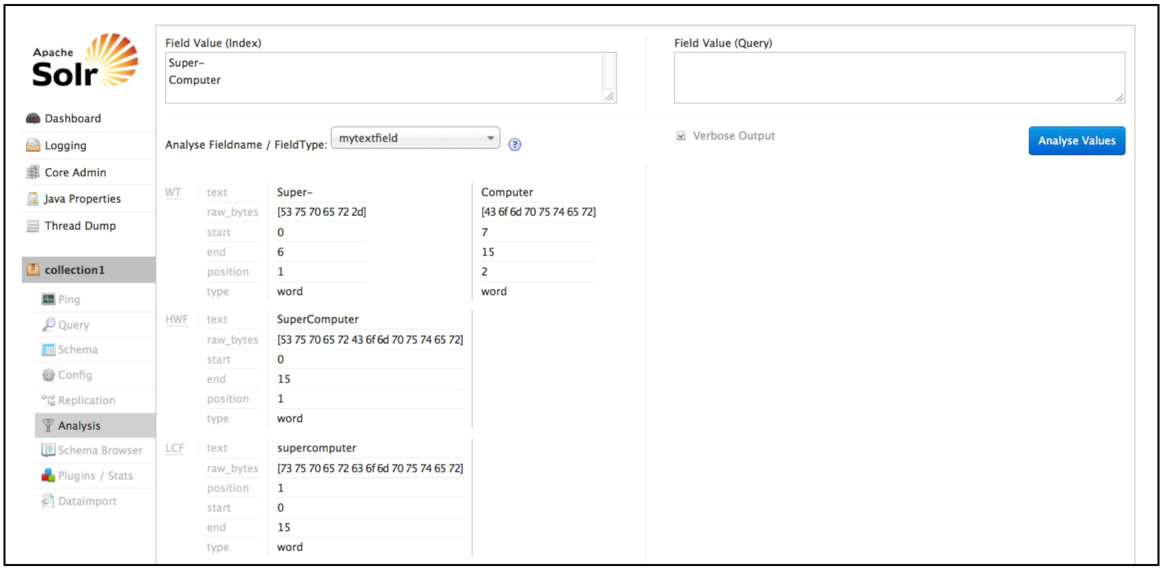
*Running index-time analyzer, verbose output.*

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`, 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 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>
```

Re-submitting the form by clicking "Analyse Values" again, we see the result in the screen shot below.



Field Value (Index)

Super-
Computer

Field Value (Query)

Analyse Fieldname / FieldType: mytextfield

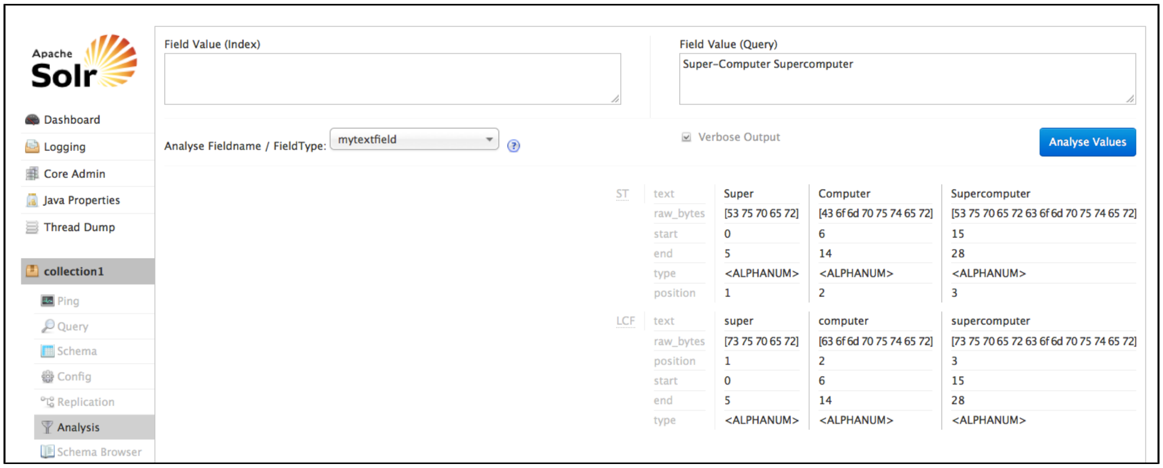
☑ Verbose Output

| | | | |
|---|---|---|---|
| WT | text | Super- | Computer |
| | raw_bytes | [53 75 70 65 72 2d] | [43 6f 6d 70 75 74 65 72] |
| | start | 0 | 7 |
| | end | 6 | 15 |
| | position | 1 | 2 |
| | type | word | word |
| HWF | text | SuperComputer | |
| | raw_bytes | [53 75 70 65 72 43 6f 6d 70 75 74 65 72] | |
| | start | 0 | |
| | end | 15 | |
| | position | 1 | |
| | type | word | |
| LCF | text | supercomputer | |
| | raw_bytes | [73 75 70 65 72 63 6f 6d 70 75 74 65 72] | |
| | position | 1 | |
| | start | 0 | |
| | end | 15 | |
| | type | word | |

*Using WhitespaceTokenizer, expected results.*

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 these results:



Field Value (Index)

Field Value (Query)

Super-Computer Supercomputer

Analyse Fieldname / FieldType: mytextfield

☑ Verbose Output

| | | | | |
|---|---|---|---|---|
| ST | text | Super | Computer | Supercomputer |
| | raw_bytes | [53 75 70 65 72] | [43 6f 6d 70 75 74 65 72] | [53 75 70 65 72 63 6f 6d 70 75 74 65 72] |
| | start | 0 | 6 | 15 |
| | end | 5 | 14 | 28 |
| | type | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> |
| | position | 1 | 2 | 3 |
| LCF | text | super | computer | supercomputer |
| | raw_bytes | [73 75 70 65 72] | [63 6f 6d 70 75 74 65 72] | [73 75 70 65 72 63 6f 6d 70 75 74 65 72] |
| | position | 1 | 2 | 3 |
| | start | 0 | 6 | 15 |
| | end | 5 | 14 | 28 |
| | type | <ALPHANUM> | <ALPHANUM> | <ALPHANUM> |

*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.

# Indexing and Basic Data Operations

This section describes how Solr adds data to its index. It covers the following topics:

- **Introduction to Solr Indexing**: An overview of Solr's indexing process.
- **Simple Post Tool**: Information about using `post.jar` to quickly upload some content to your system.
- **Uploading Data with Index Handlers**: Information about using Solr's Index Handlers to upload XML/XSLT, JSON and CSV data.
- **Uploading Data with Solr Cell using Apache Tika**: Information about using the Solr Cell framework to upload data for indexing.
- **Uploading Structured Data Store Data with the Data Import Handler**: Information about uploading and indexing data from a structured data store.
- **Updating Parts of Documents**: Information about how to use atomic updates and optimistic concurrency with Solr.
- **Detecting Languages During Indexing**: Information about using language identification during the indexing process.
- **De-Duplication**: Information about configuring Solr to mark duplicate documents as they are indexed.
- **Content Streams**: Information about streaming content to Solr Request Handlers.
- **UIMA Integration**: Information about integrating Solr with Apache's Unstructured Information Management Architecture (UIMA). UIMA lets you define custom pipelines of Analysis Engines that incrementally add metadata to your documents as annotations.

### Indexing Using Client APIs

Using client APIs, such as SolrJ, from your applications is an important option for updating Solr indexes. See the Client APIs section for more information.

## Introduction to Solr Indexing

This section 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 value (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 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 Client APIs. 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 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 Documents, Fields, and Schema Design), if one matching the field name exists.

For more information on indexing in Solr, see the Solr Wiki.

### The Solr Example Directory

The `example/` directory includes a sample Solr implementation, along with sample documents for uploading into an index. You will find the example docs in `$SOLR_HOME/example/exampledocs`.

### The `curl` Utility for Transferring Files

Many of the instructions and examples in this section 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.

> ⚠ 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 will achieve better performance with Solr Cell or the other methods described in this section.
>
> Instead of `curl`, you can use utilities such as GNU `wget` (http://www.gnu.org/software/wget/) or manage GETs and POSTS with Perl, although the command line options will differ.

## Simple Post Tool

Solr includes a simple command line tool for POSTing raw XML to a Solr port. XML data can be read from files specified as command line arguments, as raw commandline argument strings, or via STDIN.

The tool is called `post.jar` and is found in the 'exampledocs' directory: `$SOLR/example/exampledocs/post.jar` includes a cross-platform Java tool for POST-ing XML documents.

To run it, open a window and enter:

```
java -jar post.jar <list of files with  messages>
```

By default, this will contact the server at `localhost:8983`. The '-help' (or simply '-h' option will output information on its usage (i.e., `java -jar post.jar -help`.

### Using the Simple Post Tool

Options controlled by System Properties include the Solr URL to post to, the `Content-Type` of the data, whether a commit or optimize should be executed, and whether the response should be written to `STDOUT`. You may override any other request parameter through the `-Dparams` property.

This table lists the supported system properties and their defaults:

| Parameter | Values | Default | Description |
|---|---|---|---|

| | | | |
|---|---|---|---|
| -Ddata | args, stdin, files, web | files | Use **args** to pass arguments along the command line (such as a command to delete a document). Use **files** to pass a filename or regex pattern indicating paths and filenames. Use **stdin** to use standard input. Use **web** for a very simple web crawler (arguments for this would be the URL to crawl). |
| -Dtype | <content-type> | application/xml | Defines the content-type, if `-Dauto` is not used. |
| -Durl | <solr-update-url> | http://localhost:8983/solr/update | The Solr URL to send the updates to. |
| -Dauto | yes, no | no | If yes, the tool will guess the file type from file name suffix, and set type and url accordingly. It also sets the ID and file name automatically. |
| -Drecursive | yes, no | no | Will recurse into sub-folders and index all files. |
| -Dfiletypes | <type>[,<type>,..] | xml, json, csv, pdf, doc, docx, ppt, pptx, xls, xlsx, odt, odp, ods, rtf, htm, html | Specifies the file types to consider when indexing folders. |
| -Dparams | "<key>=<value>[&<key>=<value>...]" | none | HTTP GET params to add to the request, so you don't need to write the whole URL again. Values must be URL-encoded. |
| -Dcommit | yes, no | yes | Perform a commit after adding the documents. |
| -Doptimize | yes, no | no | Perform an optimize after adding the documents. |
| -Dout | yes, no | no | Write the response to an output file. |

#### Examples

There are several ways to use `post.jar`. Here are a few examples:

Add all documents with file extension `.xml`.

```
java -jar post.jar *.xml
```

Send XML arguments to delete a document from the index.

```
java -Ddata=args -jar post.jar '<delete><id>42</id></delete>'
```

Index all CSV files.

```
java -Dtype=text/csv -jar post.jar *.csv
```

Index all JSON files.

```
java -Dtype=application/json -jar post.jar *.json
```

Use the extracting request handler to index a PDF file.

```
java -Durl=[http://localhost:8983/solr/update/extract] -Dparams=literal.id=a
-Dtype=application/pdf -jar post.jar a.pdf
```

Automatically detect the content type based on the file extension.

```
java -Dauto=yes -jar post.jar a.pdf
```

Automatically detect content types in a folder, and recursively scan it for documents.

```
java -Dauto=yes -Drecursive=yes -jar post.jar afolder
```

Automatically detect content types in a folder, but limit it to PPT and HTML files.

```
java -Dauto=yes -Dfiletypes=ppt,html -jar post.jar afolder
```

## Uploading Data with Index Handlers

Index Handlers are Request Handlers designed to add, delete and update documents to the index. In addition to having plugins for importing rich documents using Tika or from structured data sources using the Data Import Handler, Solr natively supports indexing structured documents in XML, CSV and JSON.

The recommended way to configure & use request handlers is with path based names, that map to paths in the request url - but request handlers can also be specified with the `qt`(query type) parameter if the `requestDispatcher` is apprpriately configured.

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 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.

Topics covered in this section:

## The Combined UpdateRequestHandler

Prior to Solr 4, uploading content with an update request handler required declaring a unique request handler for the format of the content in the request. Now, there is a unified update request handler that supports XML, CSV, JSON, and javabin update requests, delegating to the appropriate `ContentStreamLoader` based on the `Content-Type` of the `ContentStream`.

### UpdateRequestHandler Configuration

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

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

### XML Formatted Index Updates

Index update commands can be sent as XML message to the update handler using `Content-type: application/xml` or `Content-type: text/xml`.

#### *Adding Documents*

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

- The `<add>` element introduces one 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>
```

Each element has certain optional attributes which may be specified.

| Command | Optional Parameter | Parameter Description |
|---|---|---|
| <add> | commitWithin=*number* | Add the document within the specified number of milliseconds |
| <add> | overwrite=*boolean* | Default is true. Indicates if the unique key constraints should be checked to overwrite previous versions of the same document (see below) |
| <doc> | boost=*float* | Default is 1.0. Sets a boost value for the document.To learn more about boosting, see Searching. |
| <field> | boost=*float* | Default is 1.0. Sets a boost value for the field. |

If the document schema defines a unique key, then by default an `/update` operation to add a document will overwrite (ie: replace) any document in the index with the same unique key. If no unique key has been defined, indexing performance is somewhat faster, as no check has to be made for an existing documents to replace.

If you have a unique key field, but you feel confident that you can safely bypass the uniqueness check (eg: you build your indexes in batch, and your indexing code guarantees it never adds the same document more then once) you can specify the {{overwrite="false"} option when adding your documents.

### *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 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 |
|---|---|
| waitSearcher | Default is true. Blocks until a new searcher is opened and registered as the main query searcher, making the changes visible. |
| expungeDeletes | (commit only) Default is false. Merges segments that have more than 10% deleted docs, expunging them in the process. |
| maxSegments | (optimize only) Default is 1. Merges the segments down to no more than this number of segments. |

Here are examples of <commit> and <optimize> using optional attributes:

```
<commit waitSearcher="false"/>
<commit waitSearcher="false" expungeDeletes="true"/>
<optimize waitSearcher="false"/>
```

### *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, although `commitWithin` is ignored for a Delete by 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>
```

### *Rollback Operations*

The rollback command rolls back all add and deletes made to the index since the last commit. It neither calls any event listeners nor creates a new searcher. Its syntax is simple: `<rollback/>`.

### *Using `curl` to Perform Updates with the Update Request Handler.*

You can use the `curl` utility 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
```

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.

#### *Using XSLT to Transform XML Index Updates*

The UpdateRequestHandler allows you to index any arbitrary XML using the `<tr>` parameter to apply an XSL transformation. You must have an XSLT stylesheet in the solr/conf/xslt directory that can transform the incoming data to the expected `<add><doc/></add>` format, and use the `tr` parameter to specify the name of that stylesheet.

Here is an example XSLT stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <add>
      <xsl:apply-templates select="/random/document"/>
    </add>
  </xsl:template>

  <xsl:template match="document">

    <doc boost="5.5">
      <xsl:apply-templates select="*"/>
    </doc>
  </xsl:template>

  <xsl:template match="node">
    <field name="{@name}">
      <xsl:if test="@enhance!=''">
        <xsl:attribute name="boost"><xsl:value-of select="@enhance"/></xsl:attribute>
      </xsl:if>
      <xsl:value-of select="@value"/>
    </field>
  </xsl:template>

</xsl:stylesheet>
```

This stylesheet transforms Solr's XML search result format into Solr's Update XML syntax. One example is to copy a Solr1.3 index (which does not have CSV response writer) into a format which can be indexed into another Solr file (provided that all fields are stored):

```
http://localhost:8983/solr/select?q=*:*&wt=xslt&tr=updateXml.xsl&rows=1000
```

You can also use the stylesheet in `XsltUpdateRequestHandler` to transform an index when updating:

```
curl "http://localhost:8983/solr/update?commit=true&tr=updateXml.xsl" -H
"Content-Type: text/xml" --data-binary @myexporteddata.xml
```

For more information about the XML Update Request Handler, see https://wiki.apache.org/solr/UpdateXmlMessages .

### JSON Formatted Index Updates

JSON formatted update requests may be sent to Solr's `/update` handler using the `Content-Type` "`application/json`" or "`text/json`".

JSON formatted updates can take 3 basic forms, described in depth below:

- A sequence of update commands, expressed as a top level JSON Object (aka: Map)
- A list of documents to add, expressed as a top level JSON Array containing a JSON Object per document
- A single document to add, expressed as a top level JSON Object – to differentiate this from a set of commands, the `json.command=false` request parameter is required.

#### *Adding a Single JSON Document*

The simplest way to add Documents via JSON is to send each document individually as a JSON Object, using the `j` `son.command=false` request parameter:

```
curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/collection1/update?json.command=false' --data-binary '
{
  "id": "1",
  "title": "Doc 1"
}'
```

### *Adding Multiple JSON Documents*

Adding multiple documents at one time via JSON can be done via a JSON Array of JSON Objects, where each object represents a document:

```
curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/collection1/update' --data-binary '
[
  {
    "id": "1",
    "title": "Doc 1"
  },
  {
    "id": "2",
    "title": "Doc 2"
  }
]'
```

A sample JSON file is provided at `example/exampledocs/books.json` that you can use to add some documents to the Solr example server using an Array of objects:

```
cd example/exampledocs
curl 'http://localhost:8983/solr/collection1/update?commit=true'
 --data-binary @books.json -H 'Content-type:application/json'
```

### *Sending Arbitrary JSON Update Commands*

In general, the JSON update syntax supports accepts all of the update commands that the XML update handler supports, through a straightforward mapping. Multiple commands, adding and deleting documents, may be contained in one message:

```
curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/collection1/update' --data-binary '
{
  "add": {
    "doc": {
      "id": "DOC1",
      "my_boosted_field": {        /* use a map with boost/value for a boosted field
*/
        "boost": 2.3,
        "value": "test"
      },
      "my_multivalued_field": [ "aaa", "bbb" ]   /* Can use an array for a
multi-valued field */
    }
  },
  "add": {
    "commitWithin": 5000,          /* commit this document within 5 seconds */
    "overwrite": false,            /* don't check for existing documents with the same
uniqueKey */
    "boost": 3.45,                 /* a document boost */
    "doc": {
      "f1": "v1",                  /* Can use repeated keys for a multi-valued field
*/
      "f1": "v2"
    }
  },

  "commit": {},
  "optimize": { "waitSearcher":false },

  "delete": { "id":"ID" },         /* delete by ID */
  "delete": { "query":"QUERY" }    /* delete by query */
}'
```

⚠ Comments are not allowed in JSON, but duplicate names are.

The comments in the above example are for illustrative purposes only, and can not be included in actual commands sent to Solr.

As with other update handlers, parameters such as `commit`, `commitWithin`, `optimize`, and `overwrite` may be specified in the URL instead of in the body of the message.

The JSON update format allows for a simple delete-by-id. The value of a `delete` can be an array which contains a list of zero or more specific document id's (not a range) to be deleted. For example:

```
{ "delete":"myid" }
```

```
{ "delete":["id1","id2"] }
```

The value of a "delete" can be an array which contains a list of zero or more id's to be deleted. It is not a range (start and end).

You can also specify `_version_` with each "delete":

```
{
  "delete":"id":50,
  "_version_":12345
}
```

You can specify the version of deletes in the body of the update request as well.

#### *JSON Update Convenience Paths*

In addition to the `/update` handler, there are a few additional JSON specific request handler paths available by default in Solr, that implicitly override the behavior of some request parameters:

| Path | Default Parameters |
|---|---|
| `/update/json` | `stream.contentType=application/json` |
| `/update/json/docs` | `stream.contentType=application/json`<br>`json.command=false` |

The `/update/json` path may be useful for clients sending in JSON formatted update commands from applications where setting the Content-Type proves difficult, while the `/update/json/docs` path can be particularly convenient for clients that always want to send in documents – either individually or as a list – with out needing to worry about the full JSON command syntax.

For more information about the JSON Update Request Handler, see https://wiki.apache.org/solr/UpdateJSON.

### CSV Formatted Index Updates

CSV formatted update requests may be sent to Solr's `/update` handler using `Content-Type` "`application/csv`" or "`text/csv`".

A sample CSV file is provided at `example/exampledocs/books.csv` that you can use to add some documents to the Solr example server:

```
cd example/exampledocs
curl 'http://localhost:8983/solr/collection1/update?commit=true'
 --data-binary @books.csv -H 'Content-type:application/csv'
```

#### *CSV Update 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 (g) or Per Field (f) | 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<br>trim=false |
| header | Set to true if first line of input contains field names. These will be used if the **field_name** parameter is absent. | g | |
| field_name | Comma separated list of field names to use when adding documents. | g | field_name=isbn,price,title |
| literal.<field_name> | Comma separated list of field names to use when processing literal values. | g | literal.color=red,blue,black |
| 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 |
| 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<br>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 | |

| | | | |
|---|---|---|---|
| commitWithin | Add the document within the specified number of milliseconds. | g | commitWithin=10000 |
| rowid | Map the rowid (line number) to a field specified by the value of the parameter, for instance if your CSV doesn't have a unique key and you want to use the row id as such. | g | rowid=id |
| rowidOffset | Add the given offset (as an int) to the rowid before adding it to the document. Default is 0 | g | rowidOffset=10 |

### *CSV Update Convenience Paths*

In addition to the `/update` handler, there is an additional CSV specific request handler path available by default in Solr, that implicitly override the behavior of some request parameters:

| Path | Default Parameters |
|---|---|
| `/update/csv` | `stream.contentType=application/csv` |

The `/update/csv` path may be useful for clients sending in CSV formatted update commands from applications where setting the Content-Type proves difficult.

For more information on the CSV Update Request Handler, see https://wiki.apache.org/solr/UpdateCSV.

### Nested Child Documents

Solr nested documents using a "Block Join" when indexing as a way to model documents containing other documents, such as a blog post parent document and comments as child documents -- or products as parent documents and sizes, colors, or other variations as child documents.  At query time, the Block Join Query Parsers c an be used search against these relationships. In terms of performance, indexing the relationships between documents may be more efficient than attempting to do joins only at query time, since the relationships are already stored in the index and do not need to be computed.

Nested documents may be indexed via either the XML or JSON data syntax (or using SolrJ) - but regardless of syntax, you must include a field that identifies the parent document as a parent; it can be any field that suits this purpose, and it will be used as input for the block join query parsers.

### *XML Examples*

For example, here are two documents and their child documents:

```
<add>
  <doc>
  <field name="id">1</field>
  <field name="title">Solr adds block join support</field>
  <field name="content_type">parentDocument</field>
    <doc>
     <field name="id">2</field>
     <field name="comments">SolrCloud supports it too!</field>
    </doc>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Lucene and Solr 4.5 is out</field>
    <field name="content_type">parentDocument</field>
    <doc>
     <field name="id">4</field>
     <field name="comments">Lots of new features</field>
    </doc>
  </doc>
</add>
```

In this example, we have indexed the parent documents with the field `content_type`, which has the value "parentDocument". We could have also used a boolean field, such as `isParent`, with a value of "true", or any other similar approach.

#### *JSON Examples*

This example is equivalent to the XML example above, note the special `_childDocuments_` key need to indicate the nested documents in JSON.

```
[
  {
    "id": "1",
    "title": "Solr adds block join support",
    "content_type": "parentDocument",
    "_childDocuments_": [
      {
        "id": "2",
        "comments": "SolrCloud supports it too!"
      }
    ]
  },
  {
    "id": "3",
    "title": "Lucene and Solr 4.5 is out",
    "content_type": "parentDocument",
    "_childDocuments_": [
      {
        "id": "4",
        "comments": "Lots of new features"
      }
    ]
  }
]
```

### *Transforming and Indexing custom JSON data*

This helps index JSON into a valid Solr document according to the users configuration. It lets the user to split a single JSON file into 1 or more Solr documents. The final indexed document can be controlled using the mapping passed along the request . One or more valid JSON documents can be sent to the /update/json/docs path with the configuration params.

#### Mapping params

- **split** : This parameter is required if you wish to transform the input JSON . This is the path at which the JSON must be split . If the entire JSON makes a single solr document , the path must be “/” .
- **f** : This is a multivalued mapping parameter . At least one field mapping must be provided . The format of the parameter is {target-field-name}:{json-path} . The ‘json-path’ is a required part . target-field-name is the name of the field in the input Solr document.  It is optional and it is automatically derived from the input json
- **echo** : This is for debugging purpose only . set it to true , if you want the docs to be returned as a response. Nothing will be indexed

example 1:

```
curl 'http://localhost:8983/solr/collection1/update/json/docs'\
'?split=/exams'\
'&f=first:/first'\
'&f=last:/last'\
'&f=grade:/grade'\
'&f=subject:/exams/subject'\
'&f=test:/exams/test'\
'&f=marks:/exams/marks'\
 -H 'Content-type:application/json' -d '
{
  "first": "John",
  "last": "Doe",
  "grade": 8,
  "exams": [
      {
        "subject": "Maths",
        "test"   : "term1",
        "marks"  : 90},
      {
        "subject": "Biology",
        "test"   : "term1",
        "marks"  : 86}
  ]
}'
```

This indexes the following two docs

```
{
  "first":"John",
  "last":"Doe",
  "marks":90,
  "test":"term1",
  "subject":"Maths",
  "grade":8
}
{
  "first":"John",
  "last":"Doe",
  "marks":86,
  "test":"term1",
  "subject":"Biology",
  "grade":8
}
```

As the final field names are the same as the input document fields, the request can be simplified as,

example 2 :

```
curl 'http://localhost:8983/solr/collection1/update/json/docs'\
'?split=/exams'\
'&f=/first'\
'&f=/last'\
'&f=/grade'\
'&f=/exams/subject'\
'&f=/exams/test'\
'&f=/exams/marks'\
 -H 'Content-type:application/json' -d '
{
  "first": "John",
  "last": "Doe",
  "grade": 8,
  "exams": [
      {
        "subject": "Maths",
        "test"   : "term1",
        "marks"  : 90},
      {
        "subject": "Biology",
        "test"   : "term1",
        "marks"  : 86}
  ]
}'
```

### Wildcards

Instead of specifying all the field names explicitly, it is possible to specify wildcards to map fields automatically. There are two restrictions: wildcards can only be used at the end of the json-path; and the split path cannot use wildcards. A single asterisk "*" maps only to direct children, and a double asterisk "**" maps recursively to all descendants. The following are example wildcard path mappings:

- f=/docs/* : maps all the fields under docs and in the name as given in json
- f=/docs/** : maps all the fields under docs and its children in the name as given in json
- f=searchField:/docs/* : maps all fields under /docs to a single field called ‘searchField’
- f=searchField:/docs/** : maps all fields under /docs and its children to searchField

With wildcards we can simplify our previous example as follows

example 3:

```
curl 'http://localhost:8983/solr/collection1/update/json/docs'\
'?split=/exams'\
'&f=/**'\
 -H 'Content-type:application/json' -d '
{
  "first": "John",
  "last": "Doe",
  "grade": 8,
  "exams": [
      {
        "subject": "Maths",
        "test"   : "term1",
        "marks"  : 90},
      {
        "subject": "Biology",
        "test"   : "term1",
        "marks"  : 86}
  ]
}'
```

It is also possible to send all the values to a single field and do a full text search on that . This is a good option to blindly index and query JSON documents without worrying about fields and schema

example 4 :

```
curl 'http://localhost:8983/solr/collection1/update/json/docs'\
'?split=/'\
'&f=txt:/**'\
 -H 'Content-type:application/json' -d '
{
  "first": "John",
  "last": "Doe",
  "grade": 8,
  "exams": [
      {
        "subject": "Maths",
        "test"   : "term1",
        "marks"  : 90},
      {
        "subject": "Biology",
        "test"   : "term1",
        "marks"  : 86}
  ]
}'
```

## Uploading Data with Solr Cell using Apache Tika

Topics covered in this section:

Solr uses code from the Apache Tika project to provide a framework for incorporating many different file-format parsers such as Apache PDFBox and Apache POI 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.

> As of version 4.8, Solr uses Apache Tika v1.5.

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.

If you want to supply your own `ContentHandler` for Solr to use, you can extend the `ExtractingRequestHandler` and override the `createFactory()` method. This factory is responsible for constructing the `SolrContentHandler` that interacts with Tika, and allows literals to override Tika-parsed values. Set the parameter `literalsOverride`, which normally defaults to *true, to *false to append Tika-parsed values to literal values.

For more information on Solr's Extracting Request Handler, see https://wiki.apache.org/solr/ExtractingRequestHandler.

### Key Concepts

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

- Tika will automatically attempt to determine the input document type (Word, PDF, HTML) 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 ContentHandler. SAX is a common interface implemented for many different XML parsers. For more information, see http://www.saxproject.org/quickstart.html.
- 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://tika.apache.org/1.5/formats.html for the file types supported.
- Tika adds all the extracted text to the `content` field. This field is defined as "stored" in `schema.xml`. It is also copied to the `text` field with a `copyField` rule.
- You can map Tika's metadata fields to Solr fields. You can also boost these fields.
- You can pass in literals for field values. Literals will override Tika-parsed values, including fields in the Tika metadata object, the Tika content field, and any "captured content" fields.
- You can apply an XPath expression to the Tika XHTML to restrict the content that is produced.

While Apache Tika is quite powerful, it is not perfect and fails on some files. PDF files are particularly problematic, mostly due to the PDF format itself. In case of a failure processing any file, the `ExtractingRequestHandler` does not have a secondary mechanism to try to extract some text from the file; it will throw an exception and fail.

### Trying out Tika with the Solr Example Directory

You can try out the Tika framework using the example application included in Solr.

Start the Solr example server:

```
cd example -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:

```
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 (for example, `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

### Input Parameters

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

| Parameter | Description |
|---|---|
| boost.<*fieldname*> | Boosts the specified field by the defined float amount. (Boosting a field alters its importance in a query response. To learn about boosting fields, see Searching.) |
| capture | 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 | Indexes attributes of the Tika XHTML elements into separate fields, named after the element. If set to true, for example, when extracting from HTML, Tika can return the href attributes in <a> tags as fields named "a". See the examples below. |
| commitWithin | Add the document within the specified number of milliseconds. |
| date.formats | Defines the date format patterns to identify in the documents. |
| defaultField | If the uprefix parameter (see below) is not specified and a field cannot be determined, the default field will be used. |
| extractOnly | 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.For an example, see http://wiki.apache.org/solr/TikaExtractOnlyExampleOutput. |
| extractFormat | Default is "xml", but the other option is "text". 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*> | Maps (moves) one field name to another. The `source_field` must be a field in incoming documents, and the value is the Solr field to map to. Example: `fmap.content=text` causes the data in the `content` field generated by Tika to be moved to the Solr's `text` field. |
| literal.<*fieldname*> | Populates a field with the name supplied with the specified value for each document. The data can be multivalued if the field is multivalued. |
| literalsOverride | If true (the default), literal field values will override other values with the same field name. If false, literal values defined with `literal.<fieldname>` will be appended to data already in the fields extracted from Tika. If setting `literalsOverride` to "false", the field must be multivalued. |

| | |
|---|---|
| lowernames | Values are "true" or "false". If true, all field names will be mapped to lowercase with underscores, if needed. For example, "Content-Type" would be mapped to "content_type." |
| multipartUploadLimitInKB | Useful if uploading very large documents, this defines the KB size of documents to allow. |
| passwordsFile | Defines a file path and name for a file of file name to password mappings. |
| resource.name | Specifies the optional name of the file. Tika can use it as a hint for detecting a file's MIME type. |
| resource.password | Defines a password to use for a password-protected PDF or OOXML file |
| tika.config | Defines a file path and name to a customized Tika configuration file. This is only required if you have customized your Tika implementation. |
| uprefix | 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 | When extracting, only return Tika XHTML content that satisfies the given XPath expression. See http://tika.apache.org/1.5/index.html for details on the format of Tika XHTML. See also http://wiki.apache.org/solr/TikaExtractOnlyExampleOutput. |

### 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>`. If `literalsOverride=false`, literals will be appended as multi-value to the Tika-generated field.
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.

### 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 is 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. 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
```

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" />
  ...
```

#### Multi-Core Configuration

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

For more information about Solr cores, see The Well-Configured Solr Instance.

### Indexing Encrypted Documents with the ExtractingUpdateRequestHandler

The ExtractingRequestHandler will decrypt encrypted files and index their content if you supply a password in either `resource.password` on the request, or in a `passwordsFile` file.

In the case of `passwordsFile`, the file supplied must be formatted so there is one line per rule. Each rule contains a file name regular expression, followed by "=", then the password in clear-text. Because the passwords are in clear-text, the file should have strict access restrictions.

```
# This is a comment
myFileName = myPassword
.*\.docx$ = myWordPassword
.*\.pdf$ = myPdfPassword
```

## Examples

### 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 Content Stream as uploaded to Solr. Depending 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 section.) |
| stream_size | The size of the stream in bytes. |
| stream_content_type | The content type of the stream, if available. |

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

### Examples of Uploads Using the Extraction Request Handler

#### *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&defaultFie
ld=text&fmap.div=foo_t&capture=div" -F "tutorial=@tutorial.pdf"
```

#### *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&defaultFie
ld=text&capture=div&fmap.div=foo_t&boost.foo_t=3" -F "tutorial=@tutorial.pdf"
```

#### *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&defaultFie
ld=text&capture=div&fmap.div=foo_t&boost.foo_t=3&literal.blah_s=Bah" -F
"tutorial=@tutorial.pdf"
```

#### *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&defaultFie
ld=text&capture=div&fmap.div=foo_t&boost.foo_t=3&literal.id=id&xpath=/xhtml:html/xhtml
:body/xhtml:div/descendant:node()" -F "tutorial=@tutorial.pdf"
```

### Extracting Data without Indexing It

Solr allows you to extract data without indexing. You might want to do this if you're using Solr solely 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'
```

## 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'
```

## 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 Client APIs.

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 HttpSolrServer("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<Object> 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.

### Related Topics

- ExtractingRequestHandler

## Uploading Structured Data Store Data with the Data Import Handler

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.

> ⚠ The DataImportHandler jars are no longer included in the Solr WAR. You should add them to Solr's lib directory, or reference them via the `<lib>` directive in `solrconfig.xml`.

For more information about the Data Import Handler, see https://wiki.apache.org/solr/DataImportHandler.

Topics covered in this section:

### 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 available interface. |

### Configuration

### Configuring `solrconfig.xml`

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>
```

The only required parameter is the `config` parameter, which specifies the location of the DIH configuration file that 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.

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.

### Configuring the DIH Configuration File

There is a sample DIH application distributed with Solr 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. The sample DIH configuration can be found in `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. More information about the parameters and options shown here are described in the sections following.

```
<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" />

<!-- 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 -->
  <document>

<!-- The possible attributes for the entity element are described below.
     Entity elements may contain one or more 'field' elements, which map
     the data source field names to Solr fields, and optionally specify
     per-field transformations -->
<!-- this entity is the 'root' entity. -->
    <entity name="item" query="select * from item"
            deltaQuery="select id from item where last_modified >
'${dataimporter.last_index_time}'">
      <field column="NAME" name="name" />

<!-- This entity is nested and reflects the one-to-many relationship between an item
and its multiple features.
     Note the use of variables; ${item.ID} is the value of the column 'ID' for the
current item
     ('item' referring to the entity name)  -->
      <entity name="feature"
              query="select DESCRIPTION from FEATURE where ITEM_ID='${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"
              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=${item_category.ITEM_ID}">
        <entity name="category"
                query="select DESCRIPTION from category where ID =
'${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>
</dataConfig>
```

Datasources can still be specified in `solrconfig.xml`. These must be specified in the defaults section of the handler in `solrconfig.xml`. However, these are not parsed until the main configuration is loaded.

The entire configuration itself can be passed as a request parameter using the `dataConfig` parameter rather than using a file. When configuration errors are encountered, the error message is returned in XML format.

In Solr 4.1, a new property was added, the `propertyWriter` element, which allows defining the date format and locale for use with delta queries. It also allows customizing the name and location of the properties file.

The `reload-config` command is still supported, which is useful for validating a new configuration file, or if you want to specify a file, load it, and not have it reloaded again on import. If there is an `xml` mistake in the configuration a user-friendly message is returned in `xml` format. You can then fix the problem and do a `reload-config`.

> You can also view the DIH configuration in the Solr Admin UI. There is also an interface to import content.

### 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.<br>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.<br>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, and so on. |

#### 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. |

### Property Writer

The `propertyWriter` element defines the date format and locale for use with delta queries. It is an optional configuration. Add the element to the DIH configuration file, directly under the `dataConfig` element.

```
<propertyWriter dateFormat="yyyy-MM-dd HH:mm:ss" type="SimplePropertiesWriter"
directory="data" filename="my_dih.properties" locale="en_US" />
```

The parameters available are:

| Parameter | Description |
|---|---|
| dateFormat | A java.text.SimpleDateFormat to use when converting the date to text. The default is "yyyy-MM-dd HH:mm:ss". |
| type | The implementation class. Use `SimplePropertiesWriter` for non-SolrCloud installations. If using SolrCloud, use `ZKPropertiesWriter`. If this is not specified, it will default to the appropriate class depending on if SolrCloud mode is enabled. |
| directory | Used with the `SimplePropertiesWriter` only). The directory for the properties file. If not specified, the default is "conf". |
| filename | Used with the `SimplePropertiesWriter` only). The name of the properties file. If not specified, the default is the requestHandler name (as defined in `solrconfig.xml`, appended by ".properties" (i.e., "dataimport.properties"). |
| locale | The locale. If not defined, the ROOT locale is used. It must be specified as language-country. For example, `en-US`. |

### Data Sources

A data source specifies the origin of data and its type. Somewhat confusingly, some data sources are configured within the associated entity processor. Data sources 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 data source by writing a class that extends `org.apache.solr.handler.dataimport.DataSource`.

The mandatory attributes for a data source definition are its name and type. The name identifies the data source to an Entity element.

The types of data sources available are described below.

### ContentStreamDataSource

This takes the POST data as the data source. This can be used with any EntityProcessor that uses a `DataSource<Reader>`.

### 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 data sources, and two entities, as follows:

```
<dataSource name="a1" driver="org.hsqldb.jdbcDriver" ...  />
<dataSource name="a2" type=FieldReaderDataSource" />
<document>

  <!-- 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 in place of the "url" attribute -->

    <entity name="e2" dataSource="a2" processor="XPathEntityProcessor"
            dataField="e1.fieldToUseForXPath">

      <!-- Xpath configuration follows -->
      ...
    </entity>
  </entity>
```

The FieldReaderDataSource can take an `encoding` parameter, which will default to "UTF-8" if not specified.It must be specified as language-country. For example, `en-US`.

### FileDataSource

This can be used like an URLDataSource, but is used to fetch content from files on disk. The only difference from URLDataSource, when accessing disk files, is how a pathname is specified.

This data source accepts these optional attributes.

| Optional Attribute | Description |
|---|---|
| basePath | The base path relative to which the value is evaluated if it is not absolute. |
| encoding | Defines the character encoding to use. If not defined, UTF-8 is used. |

### JdbcDataSource

This is the default datasource. It's used with the SQLEntityProcessor. See the example in the FieldReaderDataSource section for details on configuration.

### URLDataSource

This data source is often used with XPathEntityProcessor to fetch content from an underlying `file://` or `http://`

location. 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. |

### Entity Processors

Entity processors extract data, transform it, and add it to a Solr index. Examples of entities 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 data source. Used if there are multiple data sources, specified, in which case each one must have a name. |
| name | Required. The unique name used to identify an entity. |
| 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. |
| cacheImpl | Optional. A class (which must implement `DIHCache`) to use for caching this entity when doing lookups from an entity which wraps it. Provided implementation is "`SortedMapBachedCache`". |
| cacheKey | The name of a property of this entity to use as a cache key if `cacheImpl` is specified. |
| cacheLookup | An entity + property name that will be used to lookup cached instances of this entity if `cacheImpl` is specified. |

Caching of entities in DIH is provided to avoid repeated lookups for same entities again and again. The default `SortedMapBachedCache` is a `HashMap` where a key is a field in the row and the value is a bunch of rows for that same key.

In the example below, each `manufacturer` entity is cached using the '`id`' property as a cache key.  Cache lookups will be performed for each `product` entity based on the product's "`manu`" property. When the cache has no data for a particular key, the query is run and the cache is populated

```
<entity name="product" query="select description,sku, manu from product" >
  <entity name="manufacturer" query="select id, name from manufacturer" cacheKey="id"
cacheLookup="product.manu" cacheImpl="SortedMapBackedCache"/>
</entity>
```

#### 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. |
| 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 `${dataimporter.delta.<column-name>}` which can be used in this query. For example, `select * from tbl where id=${dataimporter.delta.id}` . |

### 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>
```

#### The MailEntityProcessor

The MailEntityProcessor uses the Java Mail API to index email messages using the IMAP protocol. The MailEntityProcessor works by connecting to a specified mailbox using a username and password, fetching the email headers for each message, and then fetching the full email contents to construct a document (one document for each mail message). The `example-DIH/solr/mail/conf` directory in Solr's example directory includes an example `mail-data-config.xml` used to configure a MailEntityProcessor:

```
<dataConfig>
  <document>
      <entity processor="MailEntityProcessor"
              user="email@gmail.com"
              password="password"
              host="imap.gmail.com"
              protocol="imaps"
              fetchMailsSince="2009-09-20 00:00:00"
              batchSize="20"
              folders="inbox"
              processAttachement="false"
              name="sample_entity"/>
  </document>
</dataConfig>
```

The entity attributes unique to the MailEntityProcessor are shown below.

| Attribute | Use |
|---|---|
| processor | Required. Must be set to "MailEntityProcessor". |
| user | Required. Username for authenticating to the IMAP server; this is typically the email address of the mailbox owner |
| password | Required. Password for authenticating to the IMAP server |
| host | Required. The IMAP server to connect to |
| protocol | Required. The IMAP protocol to use, valid values are: imap, imaps, gimap, and gimaps |
| fetchMailsSince | Optional. Date/time used to set a filter to import messages that occur after the specified date; expected format is: yyyy-MM-dd HH:mm:ss |
| folders | Required. Comma-delimited list of folder names to pull messages from, such as "inbox" |
| recurse | Optional (default is true). Flag to indicate if the processor should recurse all child folders when looking for messages to import |
| include | Optional. Comma-delimited list of folder patterns to include when processing folders (can be a literal value or regular expression) |
| exclude | Optional. Comma-delimited list of folder patterns to exclude when processing folders (can be a literal value or regular expression);<br>excluded folder patterns take precedent over include folder patterns. |
| processAttachement<br>or<br>processAttachments | Optional (default is true). Use Tika to process message attachments. |
| includeContent | Optional (default is true). Include the message body when constructing Solr documents for indexing |

#### *Importing New Emails Only*

After running a full import, the MailEntityProcessor keeps track of the timestamp of the previous import so that subsequent imports can use the fetchMailsSince filter to only pull new messages from the mail server. This occurs automatically using the Data Import Handler dataimport.properties file (stored in conf). For instance, if you set fetchMailsSince=2014-08-22 00:00:00 in your mail-data-config.xml, then all mail messages that occur after this date will be imported on the first run of the importer. Subsequent imports will use the date of the previous import as the fetchMailsSince filter, so that only new emails since the last import are indexed each time.

#### *GMail Extensions*

When connecting to a GMail account, you can improve the efficiency of the MailEntityProcessor by setting the protocol to **gimap** or **gimaps**. This allows the processor to send the fetchMailsSince filter to the GMail server to have the date filter applied on the server, which means the processor only receives new messages from the server. However, GMail only supports date granularity, so the server-side filter may return previously seen messages if run more than once a day.

#### The TikaEntityProcessor

The TikaEntityProcessor uses Apache Tika to process incoming documents. This is similar to Uploading Data with Solr Cell using Apache Tika, but using the DataImportHandler options instead.

The `example-DIH` directory in Solr's `example` directory shows one option for using the TikaEntityProcessor. Here is the sample `data-config.xml` file:

```
<dataConfig>
  <dataSource type="BinFileDataSource" />
  <document>
    <entity name="tika-test" processor="TikaEntityProcessor"
            url="../contrib/extraction/src/test-files/extraction/solr-word.pdf"
format="text">
      <field column="Author" name="author" meta="true"/>
      <field column="title" name="title" meta="true"/>
      <field column="text" name="text"/>
    </entity>
  </document>
</dataConfig>
```

The parameters for this processor are described in the table below:

| Attribute | Use |
|---|---|
| dataSource | This parameter defines the data source and an optional name which can be referred to in later parts of the configuration if needed. This is the same dataSource explained in the description of general entity processor attributes above.<br>The available data source types for this processor are:<br>• BinURLDataSource: used for HTTP resources, but can also be used for files.<br>• BinContentStreamDataSource: used for uploading content as a stream.<br>• BinFileDataSource: used for content on the local filesystem. |
| url | The path to the source file(s), as a file path or a traditional internet URL. This parameter is required. |
| htmlMapper | Allows control of how Tika parses HTML. The "default" mapper strips much of the HTML from documents while the "identity" mapper passes all HTML as-is with no modifications. If this parameter is defined, it must be either **default** or **identity**; if it is absent, "default" is assumed. |
| format | The output format. The options are **text**, **xml**, **html** or **none**. The default is "text" if not defined. The format "none" can be used if metadata only should be indexed and not the body of the documents. |
| parser | The default parser is `org.apache.tika.parser.AutoDetectParser`. If a custom or other parser should be used, it should be entered as a fully-qualified name of the class and path. |
| fields | The list of fields from the input documents and how they should be mapped to Solr fields. If the attribute `meta` is defined as "true", the field will be obtained from the metadata of the document and not parsed from the body of the main text. |

#### The FileListEntityProcessor

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: `fileAbsolutePa` `th`, `fileSize`, `fileLastModified`, `fileName` which can be used in the nested processor. This processor does not use a data source.

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-ddHH: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>
```

#### LineEntityProcessor

This EntityProcessor reads all content from the data source on a line by line basis and returns a field called `rawLine` 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 data source. If this value is relative and you are using FileDataSource or URLDataSource, 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 XPathEntityProcessor.

#### PlainTextEntityProcessor

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`, `URLDataSource`).

### 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 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. |
| TemplateTransformer | Transform a field using a template. |

These transformers are described below.

#### ClobTransformer

You can use the ClobTransformer 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. See http://en.wikipedia.org/wiki/Character_large_object. Here's an example of invoking the ClobTransformer.

```
<entity name="e" transformer="ClobTransformer" ...>
  <field column="hugeTextField" clob="true" />
  ...
</entity>
```

The ClobTransformer accepts these attributes:

| 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 |

#### 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. |
| locale | The locale to use for date transformations. If not specified, the ROOT locale will be used. It must be specified as language-country. For example, `en-US`. |

Here is example code that returns the date rounded up to the month "2007-JUL":

```
<entity name="en" pk="id" transformer="DateFormatTransformer" ... >
  ...
  <field column="date" sourceColName="fulldate" dateTimeFormat="yyyy-MMM"/>
</entity>
```

#### The HTMLStripTransformer

You can use this transformer to strip HTML out of a field. For example:

```
<entity name="e" transformer="HTMLStripTransformer" ... >
  <field column="htmlText" stripHTML="true" />
  ...
</entity>
```

There is one attribute for this transformer, `stripHTML`, which is a boolean value (true/false) to signal if the HTMLStripTransformer should process the field or not.

#### The LogTransformer

You can use this transformer 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.

#### The NumberFormatTransformer

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 ROOT 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>
```

#### 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 is 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 result set, 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.

#### 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 7; 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 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>
```

#### 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>
```

### 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; that is, 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. |

## Updating Parts of Documents

Once you have indexed the content you need in your Solr index, you will want to start thinking about your strategy for dealing with changes to those documents. Solr supports two approaches to updating documents that have only partially changed.

The first is *atomic updates*. This approach allows changing only one or more fields of a document without having to re-index the entire document.

The second approach is known as *optimistic concurrency* or *optimistic locking*. It is a feature of many NoSQL databases, and allows conditional updating a document based on it's version. This approach includes semantics and rules for how to deal with version matches or mis-matches.

Atomic Updates and Optimistic Concurrency may be used as independent strategies for managing changes to documents, or they may be combined: you can use optimistic concurrency to conditionally apply an atomic update.

### Atomic Updates

Solr supports several modifiers that atomically update values of a document. This allows updating only specific fields, which can help speed indexing processes in an environment where speed of index additions is critical to the application.

To use atomic updates, add a modifier to the field that needs to be updated. The content can be updated, added to, or incrementally increased if a number.

| Modifier | Usage |
|---|---|
| set | Set or replace the field value(s) with the specified value(s), or remove the values if 'null' or empty list is specified as the new value.<br>May be specified as a single value, or as a list for multivalued fields |
| add | Adds the specified values to a multivalued field.<br>May be specified as a single value, or as a list. |
| remove | Removes (all occurrences of) the specified values from a multivalued field.<br>May be specified as a single value, or as a list. |
| inc | Increments a numeric value by a specific amount.<br>Must be specified as a single numeric value. |

⚠ All original source fields must be stored for field modifiers to work correctly, which is the Solr default.

For example, if the following document exists in our collection:

```
{"id":"mydoc",
 "price":10,
 "popularity":42,
 "categories":["kids"],
 "promo_ids":["a123x"],
 "tags":["free_to_try","buy_now","clearance","on_sale"]
}
```

And we apply the following update command:

```
{"id":"mydoc",
 "price":{"set":99},
 "popularity":{"inc":20},
 "categories":{"add":["toys","games"]},
 "promo_ids":{"remove":"a123x"},
 "tags":{"remove":["free_to_try","on_sale"]}
}
```

The resulting document in our collection will be:

```
{"id":"mydoc",
 "price":999,
 "popularity":62,
 "categories":["kids","toys","games"],
 "tags":["buy_now","clearance"]
}
```

## Optimistic Concurrency

Optimistic Concurrency is a feature of Solr that can be used by client applications which update/replace documents to ensure that the document they are replacing/updating has not been concurrently modified by another client application. This feature works by requiring a `_version_` field on all documents in the index, and comparing that to a `_version_` specified as part of the update command. By default, Solr's `schema.xml` includes a `_version_` field , and this field is automatically added to each new document.

In general, using optimistic concurrency involves the following work flow:

1. A client reads a document. In Solr, one might retrieve the document with the `/get` handler to be sure to have the latest version.
2. A client changes the document locally.
3. The client resubmits the changed document to Solr, for example, perhaps with the `/update` handler.
4. If there is a version conflict (HTTP error code 409), the client starts the process over.

When the client resubmits a changed document to Solr, the `_version_` can be included with the update to invoke optimistic concurrency control. Specific semantics are used to define when the document should be updated or when to report a conflict.

- If the content in the `_version_` field is greater than '1' (i.e., '12345'), then the `_version_` in the document must match the `_version_` in the index.
- If the content in the `_version_` field is equal to '1', then the document must simply exist. In this case, no version matching occurs, but if the document does not exist, the updates will be rejected.
- If the content in the `_version_` field is less than '0' (i.e., '-1'), then the document must **not** exist. In this case,

no version matching occurs, but if the document exists, the updates will be rejected.

- If the content in the `_version_` field is equal to '0', then it doesn't matter if the versions match or if the document exists or not. If it exists, it will be overwritten; if it does not exist, it will be added.

If the document being updated does not include the `_version_` field, and atomic updates are not being used, the document will be treated by normal Solr rules, which is usually to discard it

For more information, please also see Yonik Seeley's presentation on NoSQL features in Solr 4 from Apache Lucene EuroCon 2012.

> **Power Tip**
> The `_version_` field is by default stored in the inverted index (`indexed="true"`). However, for some systems with a very large number of documents, the increase in FieldCache memory requirements may be too costly. A solution can be to declare the `_version_` field as DocValues:
>
> **Sample field definition**
>
> ```
> <field name="_version_" type="long" indexed="false" stored="true"
> required="true" docValues="true"/>
> ```

### Document Centric Versioning Constraints

Optimistic Concurrency is extremely powerful, and works very efficiently because it uses an internally assigned, globally unique values for the `_version_` field. However, In some situations users may want to configure their own document specific version field, where the version values are assigned on a per-document basis by an external system, and have Solr reject updates that attempt to replace a document with an "older" version. In situations like this the `DocBasedVersionConstraintsProcessorFactory` can be useful.

The basic usage of `DocBasedVersionConstraintsProcessorFactory` is to configure it in `solrconfig.xml` as part of the UpdateRequestProcessorChain and specify the name of the `versionField` in your schema that should be checked when validating updates:

```
<processor class="solr.DocBasedVersionConstraintsProcessorFactory">
  <str name="versionField">my_version_l</str>
</processor>
```

Once configured, this update processor will reject (HTTP error code 409) any attempt to update an existing document where the value of the `my_version_l` field in the "new" document is not greater then the value of that field in the existing document.

`DocBasedVersionConstraintsProcessorFactory` supports two additional configuration params which are optional:

- `ignoreOldUpdates` - A boolean option which defaults to `false`. If set to `true` then instead of rejecting updates where the `versionField` is too low, the update will be silently ignored (and return a status 200 to the client).
- `deleteVersionParam` - A String parameter that can be specified to indicate that this processor should also inspect Delete By Id commands. The value of this configuration option should be the name of a request

parameter that the processor will now consider mandatory for all attempts to Delete By Id, and must be be used by clients to specify a value for the `versionField` which is greater then the existing value of the document to be deleted. When using this request param, any Delete By Id command with a high enough document version number to succeed will be internally converted into an Add Document command that replaces the existing document with a new one which is empty except for the Unique Key and `versionField` to keeping a record of the deleted version so future Add Document commands will fail if their "new" version is not high enough.

Please consult the processor javadocs and test configs for additional information and example usages.

## De-Duplication

Preventing duplicate or near duplicate documents from entering an index or tagging documents with a signature/fingerprint for duplicate field collapsing can be efficiently achieved with a low collision or fuzzy hash algorithm. Solr natively supports de-duplication techniques of this type via the `<Signature>` class and allows for the easy addition of new hash/signature implementations. A Signature can be implemented several ways:

| Method | Description |
|---|---|
| MD5Signature | 128 bit hash used for exact duplicate detection. |
| Lookup3Signature | 64 bit hash used for exact duplicate detection, much faster than MD5 and smaller to index |
| TextProfileSignature | Fuzzy hashing implementation from nutch for near duplicate detection. Its tunable but works best on longer text. |

Other, more sophisticated algorithms for fuzzy/near hashing can be added later.

⚠ Adding in the deduplication process will change the `allowDups` setting so that it applies to an update Term (with `signatureField` in this case) rather than the unique field Term. Of course the `signatureField` could be the unique field, but generally you want the unique field to be unique. When a document is added, a signature will automatically be generated and attached to the document in the specified `signatureField`.

#### Configuration Options

#### *In `solrconfig.xml`*

The `SignatureUpdateProcessorFactory` has to be registered in the solrconfig.xml as part of the UpdateRequestProcessorChain:

```
<updateRequestProcessorChain name="dedupe">
  <processor class="solr.processor.SignatureUpdateProcessorFactory">
    <bool name="enabled">true</bool>
    <str name="signatureField">id</str>
    <bool name="overwriteDupes">false</bool>
    <str name="fields">name,features,cat</str>
    <str name="signatureClass">solr.processor.Lookup3Signature</str>
  </processor>
</updateRequestProcessorChain>
```

| Setting | Default | Description |
|---|---|---|

| | | |
|---|---|---|
| signatureClass | org.apache.solr.update.processor.Lookup3Signature | A Signature implementation for generating a signature hash. |
| fields | all fields | The fields to use to generate the signature hash in a comma separated list. By default, all fields on the document will be used. |
| signatureField | signatureField | The name of the field used to hold the fingerprint/signature. Be sure the field is defined in schema.xml. |
| enabled | true | Enable/disable deduplication factory processing |

#### *In `schema.xml`*

If you are using a separate field for storing the signature you must have it indexed:

```
<field name="signature" type="string" stored="true" indexed="true" multiValued="false"
/>
```

Be sure to change your update handlers to use the defined chain, i.e.

```
<requestHandler name="/update" >
  <lst name="defaults">
    <str name="update.chain">dedupe</str>
  </lst>
</requestHandler>
```

> The update processor can also be specified per request with a parameter of `update.chain=dedupe`.

## Detecting Languages During Indexing

Solr can identify languages and map text to language-specific fields during indexing using the `langid` UpdateRequestProcessor. Solr supports two implementations of this feature:

- Tika's language detection feature: http://tika.apache.org/0.10/detection.html
- LangDetect language detection: http://code.google.com/p/language-detection/

You can see a comparison between the two implementations here: http://blog.mikemccandless.com/2011/10/accuracy-and-performance-of-googles.html. In general, the LangDetect implementation supports more languages with higher performance.

For specific information on each of these language identification implementations, including a list of supported languages for each, see the relevant project websites. For more information about the `langid` UpdateRequestProcessor, see the Solr wiki: http://wiki.apache.org/solr/LanguageDetection. For more information about language analysis in Solr, see Language Analysis.

### Configuring Language Detection

You can configure the `langid` UpdateRequestProcessor in `solrconfig.xml`. Both implementations take the same parameters, which are described in the following section. At a minimum, you must specify the fields for language identification and a field for the resulting language code.

#### Configuring Tika Language Detection

Here is an example of a minimal Tika `langid` configuration in `solrconfig.xml`:

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

#### Configuring LangDetect Language Detection

Here is an example of a minimal LangDetect `langid` configuration in `solrconfig.xml`:

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

### `langid` Parameters

As previously mentioned, both implementations of the `langid` UpdateRequestProcessor take the same parameters.

| Parameter | Type | Default | Required | Description |
|---|---|---|---|---|
| langid | Boolean | true | no | Enables and disables language detection. |
| langid.fl | string | none | yes | A comma- or space-delimited list of fields to be processed by `langid`. |
| langid.langField | string | none | yes | Specifies the field for the returned language code. |
| langid.langsField | multivalued string | none | no | Specifies the field for a list of returned language codes. If you use `langid.map.individual`, each detected language will be added to this field. |
| langid.overwrite | Boolean | false | no | Specifies whether the content of the `langField` and `langsField` fields will be overwritten if they already contain values. |

|  |  |  |  |  |
|---|---|---|---|---|
| langid.lcmap | string | none | false | A space-separated list specifying colon delimited language code mappings to apply to the detected languages. For example, you might use this to map Chinese, Japanese, and Korean to a common `cjk` code, and map both American and British English to a single `en` code by using `langid.lcmap=ja:cjk zh:cjk ko:cjk en_GB:en en_US:en`. This affects both the values put into the `langField` and `langsField` fields, as well as the field suffixes when using `langid.map`, unless overridden by `langid.map.lcmap` |
| langid.threshold | float | 0.5 | no | Specifies a threshold value between 0 and 1 that the language identification score must reach before `langid` accepts it. With longer text fields, a high threshold such at 0.8 will give good results. For shorter text fields, you may need to lower the threshold for language identification, though you will be risking somewhat lower quality results. We recommend experimenting with your data to tune your results. |
| langid.whitelist | string | none | no | Specifies a list of allowed language identification codes. Use this in combination with `langid.map` to ensure that you only index documents into fields that are in your schema. |
| langid.map | Boolean | false | no | Enables field name mapping. If true, Solr will map field names for all fields listed in `langid.fl`. |
| langid.map.fl | string | none | no | A comma-separated list of fields for `langid.map` that is different than the fields specified in `langid.fl`. |
| langid.map.keepOrig | Boolean | false | no | If true, Solr will copy the field during the field name mapping process, leaving the original field in place. |
| langid.map.individual | Boolean | false | no | If true, Solr will detect and map languages for each field individually. |

| | | | | |
|---|---|---|---|---|
| langid.map.individual.fl | string | none | no | A comma-separated list of fields for use with `langid.map.individual` that is different than the fields specified in `langid.fl`. |
| langid.fallbackFields | string | none | no | If no language is detected that meets the `langid.threshold` score, or if the detected language is not on the `langid.whitelist`, this field specifies language codes to be used as fallback values. If no appropriate fallback languages are found, Solr will use the language code specified in `langid.fallback`. |
| langid.fallback | string | none | no | Specifies a language code to use if no language is detected or specified in `langid.fallbackFields`. |
| langid.map.lcmap | string | determined by `langid.lcmap` | no | A space-separated list specifying colon delimited language code mappings to use when mapping field names. For example, you might use this to make Chinese, Japanese, and Korean language fields use a common `*_cjk` suffix, and map both American and British English fields to a single `*_en` by using `langid.map.lcmap=ja:cjk zh:cjk ko:cjk en_GB:en en_US:en`. |
| langid.map.pattern | Java regular expression | none | no | By default, fields are mapped as <field>_<language>. To change this pattern, you can specify a Java regular expression in this parameter. |
| langid.map.replace | Java replace | none | no | By default, fields are mapped as <field>_<language>. To change this pattern, you can specify a Java replace in this parameter. |
| langid.enforceSchema | Boolean | true | no | If false, the `langid` processor does not validate field names against your schema. This may be useful if you plan to rename or delete fields later in the UpdateChain. |

## Content Streams

When Solr RequestHandlers 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.)

### 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 full POST body is parsed as parameters and included in the Solr parameters.
- The contents of parameter `stream.body` is passed as a stream.
- If remote streaming is enabled and URL content is called for during request handling, the contents of each `stream.url` and `stream.file` parameters are fetched and passed as a stream.

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.

### RemoteStreaming

Remote streaming lets you 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.

> ⚠ 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 allow anyone to view any file on your system.

```
<!--Make sure your system has authentication before enabling remote streaming!-->
<requestParsers enableRemoteStreaming="true" multipartUploadLimitInKB="2048" />
```

### 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 the RequestHandlers.

## UIMA Integration

You can integrate the Apache Unstructured Information Management Architecture (UIMA) with Solr. UIMA lets you define custom pipelines of Analysis Engines that incrementally add metadata to your documents as annotations.

For more information about Solr UIMA integration, see https://wiki.apache.org/solr/SolrUIMA.

### Configuring UIMA

The SolrUIMA UpdateRequestProcessor is a custom update request processor that takes documents being indexed, sends them to a UIMA pipeline, and then returns the documents enriched with the specified metadata. To configure UIMA for Solr, follow these steps:

1. Copy `solr-uima-4.x.y.jar` (under `/solr-4.x.y/dist/`) and its libraries (under `contrib/uima/lib`) to a Solr libraries directory, or set `<lib/>` tags in `solrconfig.xml` appropriately to point to those jar files:

```
<lib dir="../../contrib/uima/lib" />
<lib dir="../../dist/" regex="solr-uima-\d.*\.jar" />
```

2. Modify `schema.xml`, adding your desired metadata fields specifying proper values for type, indexed, stored, and multiValued options. For example:

```
<field name="language" type="string" indexed="true" stored="true"
required="false"/>
<field name="concept" type="string" indexed="true" stored="true"
multiValued="true" required="false"/>
<field name="sentence" type="text" indexed="true" stored="true"
multiValued="true" required="false" />
```

3. Add the following snippet to `solrconfig.xml`:

```
<updateRequestProcessorChain name="uima">
  <processor
class="org.apache.solr.uima.processor.UIMAUpdateRequestProcessorFactory">
    <lst name="uimaConfig">
      <lst name="runtimeParameters">
        <str name="keyword_apikey">VALID_ALCHEMYAPI_KEY</str>
        <str name="concept_apikey">VALID_ALCHEMYAPI_KEY</str>
        <str name="lang_apikey">VALID_ALCHEMYAPI_KEY</str>
        <str name="cat_apikey">VALID_ALCHEMYAPI_KEY</str>
        <str name="entities_apikey">VALID_ALCHEMYAPI_KEY</str>
        <str name="oc_licenseID">VALID_OPENCALAIS_KEY</str>
      </lst>
      <str
name="analysisEngine">/org/apache/uima/desc/OverridingParamsExtServicesAE.xml</st
r>
      <!-- Set to true if you want to continue indexing even if text processing
fails.
           Default is false. That is, Solr throws RuntimeException and
           never indexed documents entirely in your session. -->
      <bool name="ignoreErrors">true</bool>
      <!-- This is optional. It is used for logging when text processing fails.
           If logField is not specified, uniqueKey will be used as logField.
      <str name="logField">id</str>
      -->
      <lst name="analyzeFields">
        <bool name="merge">false</bool>
        <arr name="fields">
          <str>text</str>
        </arr>
      </lst>
      <lst name="fieldMappings">
        <lst name="type">
          <str name="name">org.apache.uima.alchemy.ts.concept.ConceptFS</str>
          <lst name="mapping">
            <str name="feature">text</str>
            <str name="field">concept</str>
          </lst>
        </lst>
        <lst name="type">
          <str name="name">org.apache.uima.alchemy.ts.language.LanguageFS</str>
          <lst name="mapping">
            <str name="feature">language</str>
            <str name="field">language</str>
          </lst>
        </lst>
        <lst name="type">
          <str name="name">org.apache.uima.SentenceAnnotation</str>
          <lst name="mapping">
            <str name="feature">coveredText</str>
            <str name="field">sentence</str>
          </lst>
        </lst>
      </lst>
    </lst>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

> ⚠ `VALID_ALCHEMYAPI_KEY` is your AlchemyAPI Access Key. You need to register an AlchemyAPI Access key to use AlchemyAPI services: http://www.alchemyapi.com/api/register.html.
>
> `VALID_OPENCALAIS_KEY` is your Calais Service Key. You need to register a Calais Service key to use the Calais services: http://www.opencalais.com/apikey.
>
> `analysisEngine` must contain an AE descriptor inside the specified path in the classpath.
>
> `analyzeFields` must contain the input fields that need to be analyzed by UIMA. If `merge=true` then their content will be merged and analyzed only once.
>
> Field mapping describes which features of which types should go in a field.

4. In your `solrconfig.xml` replace the existing default UpdateRequestHandler or create a new UpdateRequestHandler:

```
<requestHandler name="/update" class="solr.XmlUpdateRequestHandler">
  <lst name="defaults">
    <str name="update.processor">uima</str>
  </lst>
</requestHandler>
```

Once you are done with the configuration your documents will be automatically enriched with the specified fields when you index them.

# Searching

This section describes how Solr works with search requests. It covers the following topics:

- Overview of Searching in Solr: An introduction to searching with Solr.
- Velocity Search UI: A sample search UI in the example configuration using the VelocityResponseWriter.
- Relevance: Conceptual information about understanding relevance in search results.
- Query Syntax and Parsing: A brief conceptual overview of query syntax and parsing. It also contains the following sub-sections:
  - Common Query Parameters: No matter the query parser, there are several parameters that are common to all of them.
  - The Standard Query Parser: Detailed information about the standard Lucene query parser.
  - The DisMax Query Parser: Detailed information about Solr's DisMax query parser.
  - The Extended DisMax Query Parser: Detailed information about Solr's Extended DisMax (eDisMax) Query Parser.
  - Function Queries: Detailed information about parameters for generating relevancy scores using values from one or more numeric fields.
  - Local Parameters in Queries: How to add local arguments to queries.
  - Other Parsers: More parsers designed for use in specific situations.
- Faceting: Detailed information about categorizing search results based on indexed terms.
- Highlighting: Detailed information about Solr's highlighting utilities. Sub-sections cover the different types of highlighters:
  - Standard Highlighter: Uses the most sophisticated and fine-grained query representation of the three highlighters.
  - FastVector Highlighter: Optimized for term vector options on fields, and good for large
- Transforming Result Documents: Detailed information about using `DocTransformers` to add computed information to individual documents
- Suggester: Detailed information about Solr's powerful autosuggest component.
- MoreLikeThis: Detailed information about Solr's similar results query component.
- Pagination of Results: Detailed information about fetching paginated results for display in a UI, or for fetching all documents matching a query.
- Result Grouping: Detailed information about grouping results based on common field values.
- Result Clustering: Detailed information about grouping search results based on cluster analysis applied to text fields. A bit like "unsupervised" faceting.
- Spatial Search: How to use Solr's spatial search capabilities.
- The Terms Component: Detailed information about accessing indexed terms and the documents that include them.
- The Term Vector Component: How to get term information about specific documents.
- The Stats Component: How to return information from numeric fields within a document set.
- The Query Elevation Component: How to force documents to the top of the results for certain queries.
- Response Writers: Detailed information about configuring and using Solr's response writers.
- Near Real Time Searching: How to include documents in search results nearly immediately after they are indexed.

documents and multiple languages.
    - Postings Highlighter: Uses similar options as the FastVector highlighter, but is more compact and efficient.
- RealTime Get: How to get the latest version of a document without opening a searcher.

- Spell Checking: Detailed information about Solr's spelling checker.

- Query Re-Ranking: Detailed information about re-ranking top scoring documents from simple queries using more complex scores.

## Overview of Searching in Solr

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. The default query parser is the DisMax query parser. Solr also includes an earlier "standard" (Lucene) query parser, and an Extended DisMax (eDisMax) 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. The Extended DisMax query parser is an improved version of DisMax that handles the full Lucene query syntax while still tolerating syntax errors. It also includes several additional features.

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).

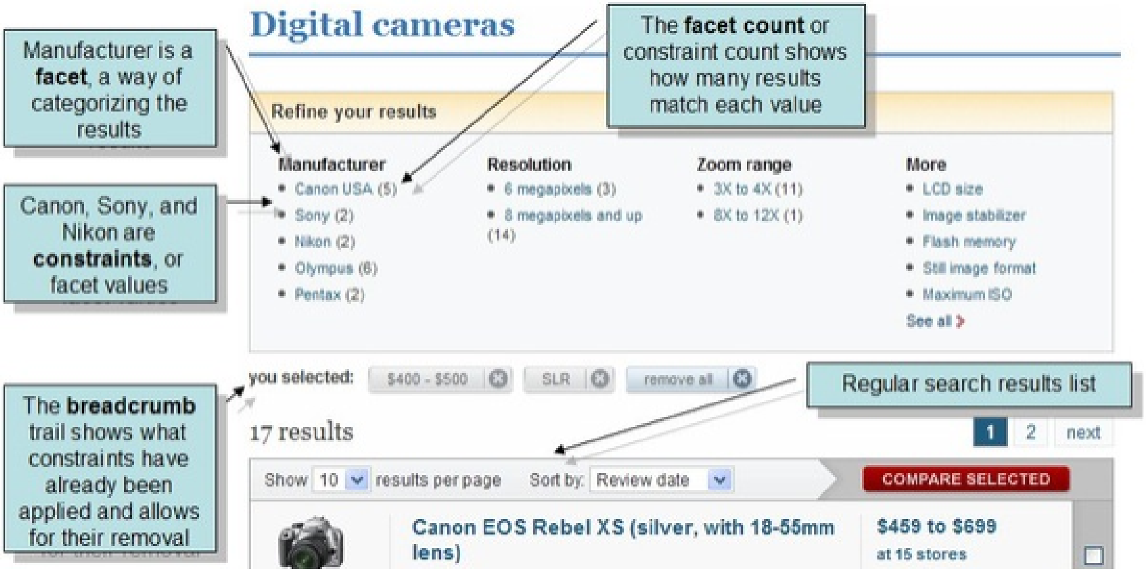
A search query can request that certain terms be highlighted in the search response; that is, the selected terms will be displayed in colored boxes so that they "jump out" on the screen of search results. **Highlighting** can make it easier to find relevant passages in long documents returned in a search. Solr supports multi-term highlighting. Solr includes a rich set of search parameters for controlling how terms are highlighted.

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. 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.



Manufacturer is a **facet**, a way of categorizing the results

Canon, Sony, and Nikon are **constraints**, or facet values

The breadcrumb trail shows what constraints have already been applied and allows for their removal

The **facet count** or constraint count shows how many results match each value

Regular search results list

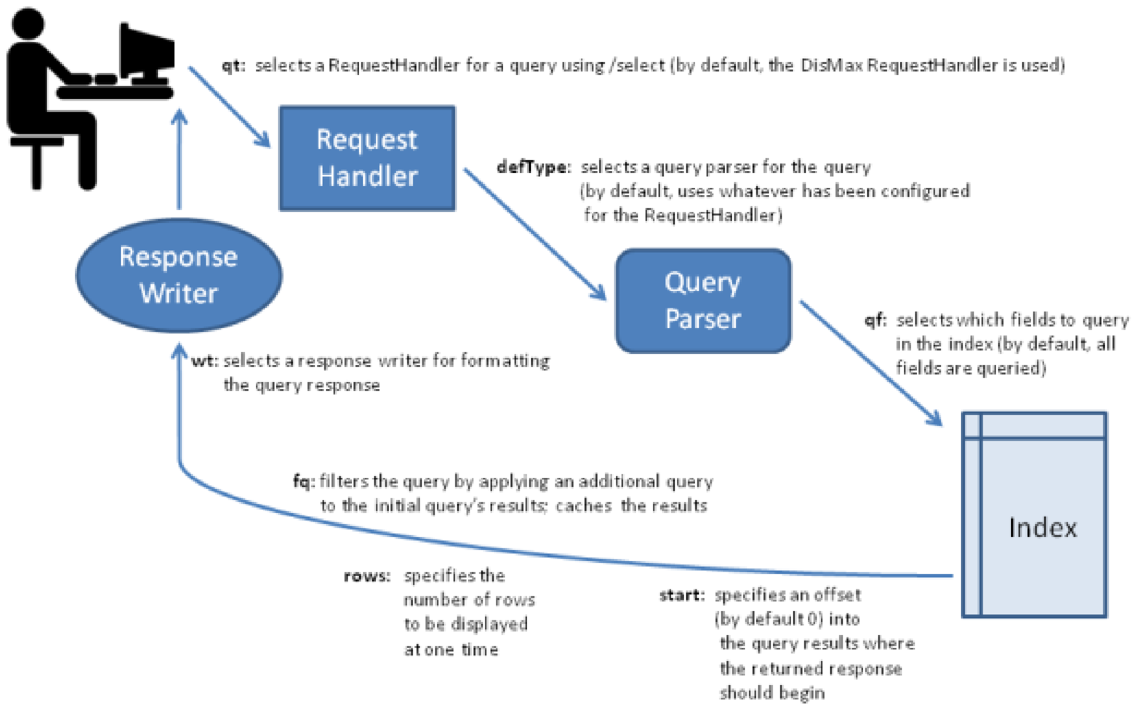
Faceting makes use of fields defined when the search applications were indexed. In the example above, these fields include categories of information that are useful for describing digital cameras: manufacturer, resolution, and zoom range.

**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.

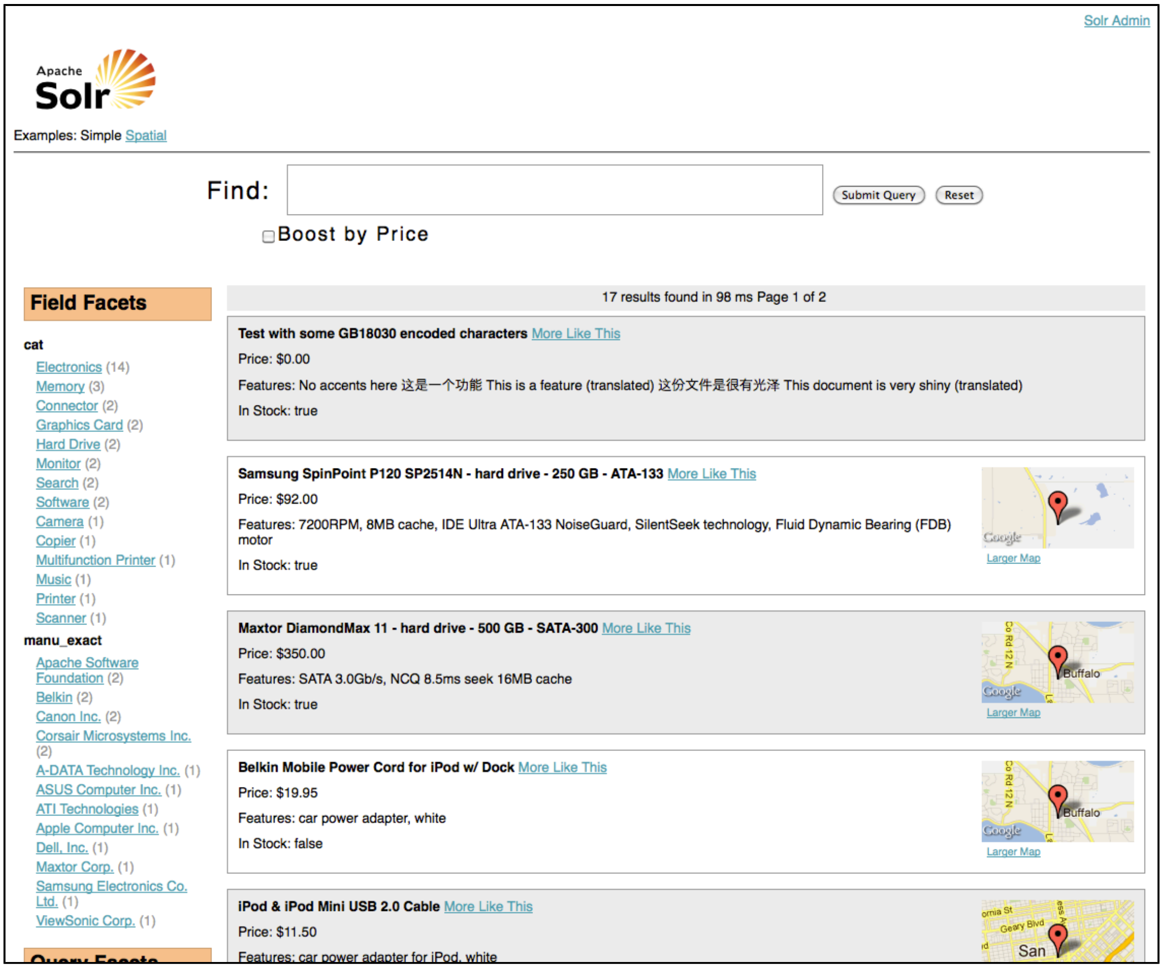
The diagram below summarizes some key elements of the search process.


qt: selects a RequestHandler for a query using /select (by default, the DisMax RequestHandler is used)
Request Handler
defType: selects a query parser for the query (by default, uses whatever has been configured for the RequestHandler)
Query Parser
qf: selects which fields to query in the index (by default, all fields are queried)
Index
fq: filters the query by applying an additional query to the initial query's results; caches the results
rows: specifies the number of rows to be displayed at one time
start: specifies an offset (by default 0) into the query results where the returned response should begin
Response Writer
wt: selects a response writer for formatting the query response

## Velocity Search UI

Solr includes a sample search UI based on the VelocityResponseWriter (also known as Solritas) that demonstrates several useful features, such as searching, faceting, highlighting, autocomplete, and geospatial searching.

You can access the Velocity sample Search UI here: `http://localhost:8983/solr/browse`



cat

Electronics (14)
Memory (3)
Connector (2)
Graphics Card (2)
Hard Drive (2)
Monitor (2)
Search (2)
Software (2)
Camera (1)
Copier (1)
Multifunction Printer (1)
Music (1)
Printer (1)
Scanner (1)

manu_exact

Apache Software Foundation (2)
Belkin (2)
Canon Inc. (2)
Corsair Microsystems Inc. (2)
A-DATA Technology Inc. (1)
ASUS Computer Inc. (1)
ATI Technologies (1)
Apple Computer Inc. (1)
Dell, Inc. (1)
Maxtor Corp. (1)
Samsung Electronics Co. Ltd. (1)
ViewSonic Corp. (1)

17 results found in 98 ms Page 1 of 2

**Test with some GB18030 encoded characters** More Like This
Price: $0.00
Features: No accents here 这是一个功能 This is a feature (translated) 这份文件是很有光泽 This document is very shiny (translated)
In Stock: true

**Samsung SpinPoint P120 SP2514N - hard drive - 250 GB - ATA-133** More Like This
Price: $92.00
Features: 7200RPM, 8MB cache, IDE Ultra ATA-133 NoiseGuard, SilentSeek technology, Fluid Dynamic Bearing (FDB) motor
In Stock: true

**Maxtor DiamondMax 11 - hard drive - 500 GB - SATA-300** More Like This
Price: $350.00
Features: SATA 3.0Gb/s, NCQ 8.5ms seek 16MB cache
In Stock: true

**Belkin Mobile Power Cord for iPod w/ Dock** More Like This
Price: $19.95
Features: car power adapter, white
In Stock: false

**iPod & iPod Mini USB 2.0 Cable** More Like This
Price: $11.50
Features: car power adapter for iPod, white

*The Velocity Search UI*

For more information about the Velocity Response Writer, see the Response Writer page.

## 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 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 Search Application Relevance Issues which is available on SearchHub.org.

## 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 main query parsers included with Solr and describes some other parsers that may be useful for particular situations. There are some query parameters common to all Solr parsers; these are discussed in the section Common Query Parameters.

The parsers discussed in this Guide are:

- The Standard Query Parser
- The DisMax Query Parser
- The Extended DisMax Query Parser
- Other Parsers

The query parser plugins are all subclasses of QParserPlugin. If you have custom parsing needs, you may want to extend that class to create your own query parser.

For more detailed information about the many query parsers available in Solr, see https://wiki.apache.org/solr/SolrQ uerySyntax.

## Common Query Parameters

The table below summarizes Solr's common query parameters, which are supported by the Standard, DisMax, and eDisMax Request Handlers.

| 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 | With version 3.6, Solr limited the query's responses to a listed set of fields. With version 4.0, this parameter returns only the score. |
| debug | Request additional debugging information in the response. Specifying the `debug=timing` parameter returns just the timing information; specifying the `debug=results` parameter returns "explain" information for each of the documents returned; specifying the `debug=query parameter` returns all of the debug information. |
| 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. |
| logParamsList | By default, Solr logs all parameters. From version 4.7, set this parameter to restrict which parameters are logged. Valid entries are the parameters to be logged, separated by commas (i.e., `logParamsList=param1,param2`). An empty list will log no parameters, so if logging all parameters is desired, do not define this additional parameter at all. |

The following sections describe these parameters in detail.

### The `defType` Parameter

The defType parameter selects the query parser that Solr should use to process the main query parameter (`q`) in the request. For example:

```
defType=dismax
```

By default, the The Standard Query Parser is used.

### 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. The directions can be entered in either all lowercase or all uppercase letters (i.e., both `asc` or `ASC`).

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 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 | 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>],...`
  - When more than one sort criteria is provided, the second entry will only be used if the first entry results in a tie. If there is a third entry, it will only be used if the first AND second entries are tied. This pattern continues with further entries.

### 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.

### 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.

### 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 The Well-Configured Solr Instance.)
- As with all parameters: special characters in an URL need to be properly escaped and encoded as hex values. Online tools are available to help you with URL-encoding. For example: http://meyerweb.com/eric/tools/dencoder/.

### 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. You can also add psuedo-fields, functions and transformers to the field list request.

This table shows some basic examples of how to use `fl`:

| 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. |

### *Function Values*

Functions can be computed for each document in the result and returned as a psuedo-field:

```
fl=id,title,product(price,popularity)
```

### *Document Transformers*

Document Transformers can be used to modify the information returned about each documents in the results of a query:

```
fl=id,title,[explain]
```

### *Field Name Aliases*

You can change the key used to in the response for a field, function, or transformer by prefixing it with a "*`displayName`*`:`". For example:

```
fl=id,sales_price:price,secret_sauce:prod(price,popularity),why_score:[explain
style=nl]
```

```
"response":{"numFound":2,"start":0,"docs":[
      {
        "id":"6H500F0",
        "secret_sauce":2100.0,
        "sales_price":350.0,
        "why_score":{
          "match":true,
          "value":1.052226,
          "description":"weight(features:cache in 2) [DefaultSimilarity], result of:",
          "details":[{
...
```

### The `debug` Parameter

In Solr 4, requesting debugging information with results has been simplified from a suite of related parameters to a single parameter that takes format information as arguments. The parameter is now simply `debug`, with the following arguments:

- `debug=true`: return debug information about the query only.
- `debug=query`: return debug information about the query only.
- `debug=timing`: return debug information about how long the query took to process.
- `debug=results`: return debug information about the results (also known as "explain")

The default behavior is not to include debugging information.

### 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.

### The `timeAllowed` Parameter

This parameter specifies the amount of time, in milliseconds, allowed for a search to complete. If this time expires before the search is complete, any partial results will be returned.

### 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.

### 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 Response Writers.

### The cache=false Parameter

Solr caches the results of all queries and filter queries by default. To disable result caching, set the `cache=false` parameter.

You can also use the `cost` option to control the order in which non-cached filter queries are evaluated. This allows you to order less expensive non-cached filters before expensive non-cached filters.

For very high cost filters, if `cache=false` and `cost>=100` and the query implements the `PostFilter` interface, a Collector will be requested from that query and used to filter documents after they have matched the main query and all other filter queries. There can be multiple post filters; they are also ordered by cost.

For example:

```
// normal function range query used as a filter, all matching documents
// generated up front and cached
fq={!frange l=10 u=100}mul(popularity,price)

// function range query run in parallel with the main query like a traditional
// lucene filter
fq={!frange l=10 u=100 cache=false}mul(popularity,price)

// function range query checked after each document that already matches the query
// and all other filters.  Good for really expensive function queries.
fq={!frange l=10 u=100 cache=false cost=100}mul(popularity,price)
```

### The `logParamsList` Parameter

By default, Solr logs all parameters of requests. From version 4.7, set this parameter to restrict which parameters of a request are logged. This may help control logging to only those parameters considered important to your organization.

For example, you could define this like:

```
logParamsList=q,fq
```

And only the 'q' and 'fq' parameters will be logged.

If no parameters should be logged, you can send `logParamsList` as empty (i.e., `logParamsList=`).

> This parameter does not only apply to query requests, but to any kind of request to Solr.

## The Standard Query Parser

Before Solr 1.3, the Standard Request Handler called the standard query parser as the default query parser. In versions since 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 is less tolerant of syntax errors than the DisMax query parser. The DisMax query parser is designed to throw as few errors as possible.

Topics covered in this section:

### 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.

### The Standard Query Parser's Response

By default, the response from the standard query parser contains one `<result>` block, which is unnamed. If the `debug` parameter is used, then an additional `<lst>` block will be returned, using the name "debug". This will contain 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.

#### *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>
```

**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).

> ⚠ 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.

***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.

***Wildcard Searches***

Solr's standard query parser supports single and multiple character wildcard searches within single terms. 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:<br>`tes*`<br>would match test, testing, and tester.<br>You can also use wildcard characters in the middle of a term. For example:<br>`te*t`<br>would match test and text.<br>`*est`<br>would match pest and test. |

As of Solr 1.4, you can use a * or ? symbol as the first character of a search with the standard query parser.

### *Fuzzy Searches*

Solr's standard query parser supports fuzzy searches based on the Damerau-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 roams, foam, & foams. It will also match the word "roam" itself.

An optional distance parameter specifies the maximum number of edits allowed, between 0 and 2, defaulting to 2. For example:

```
roam~1
```

This will match terms like roams & foam - but not foams since it has an edit distance of "2".

In many cases, stemming (reducing terms to a common stem) can produce similar effects to fuzzy searches and wildcard searches.

### *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
```

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.

#### *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.
- You can mix these types so one end of the range is inclusive and the other is exclusive. Here's an example: `count:{1 TO 10]`

#### *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).

#### 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.

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).

### 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.

When specifying Boolean operators with keywords such as AND or NOT, the keywords must appear in all uppercase.

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.

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
```

#### *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
```

> This operator is supported by both the standard query parser and the DisMax query parser.

#### *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"
```

#### *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"
```

#### *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"
```

#### *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 sub-query with two terms, escape the characters by preceding each one with a backslash:

```
\(1\+1\)\:2
```

### Grouping Terms to Form Sub-Queries

Lucene/Solr supports using parentheses to group clauses to form sub-queries. 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."

#### *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")`

### 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:
  - `_val_:myfield`
  - `_val_:"recip(rord(myfield),1,2,3)"`
- Support for any type of query parser. Prior to Solr 4.1, the "magic" field "`_query_` needed to be used to nest another query parser. However, with Solr 4.1, other query parsers can be used directly using the local parameters syntax.
  - `{!geodist d=10 p=20.5,30.2}`
- 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).

#### *Specifying Dates and Times*

Queries against fields using the `TrieDateField` type (typically range queries) should use the appropriate date syntax:

- `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]`

#### Related Topics

- Local Parameters in Queries
- Other Parsers

## 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.

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.

#### 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. If absent, defaults to `df` . |
| mm | Minimum "Should" Match: specifies a minimum number of fields that must match in a query. If no 'mm' parameter is specified in the query, or as a default in `solrconfig.xml`, the effective value of the `q.op` parameter (either in the query, as a default in `solrconfig.xml`, or from the 'defaultOperator' option in `schema.xml`) is used to influence the behavior. If `q.op` is effectively AND'ed, then mm=100%; if `q.op` is OR'ed, then mm=1. Users who want to force the legacy behavior should set a default value for the 'mm' parameter in their `solrconfig.xml` file. Users should add this as a configured default for their request handlers. This parameter tolerates miscellaneous white spaces in expressions (e.g., `" 3 < -25% 10 < -3\n", " \n-25%\n ", " \n3\n "`). |
| 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. |
| qs | Query Phrase Slop: specifies the number of positions two terms can be apart in order to match the specified phrase. Used specifically with the `qf` parameter. |
| 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 for details about function queries.) |

The sections below explain these parameters in detail.

#### *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.

⚠ The q parameter does not support wildcard characters such as *.

#### *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.

#### *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`.

#### *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%<br>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 than 1. (In other words, no matter how low or how high the calculated result, the minimum number of required matches will never be less than 1 or greater than the number of clauses.)

The default value of `mm` is 100% (meaning that all clauses must match).

#### *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.

#### *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.

#### *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.

#### *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.

#### *The `bq` (Boost Query) Parameter*

The `bq` parameter specifies an additional, optional, query clause that will be added to the user's main 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.

#### *The `bf` (Boost Functions) Parameter*

The `bf` parameter specifies functions (with optional boosts) that will be used to construct FunctionQueries which will be added to the user's main query as optional clauses that will 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 essentially just shorthand for using the `bq` param combined with the `{!func}` parser.

For example, if you want to show the most recent documents first, you could use either of the following:

```
bf=recip(rord(creationDate),1,1000,1000)
  ...or...
bq={!func}recip(rord(creationDate),1,1000,1000)
```

#### 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
```

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/?defType=dismax&q=video
```

Note that this instance is also configured with a default field list, which can be overridden in the URL.

```
http://localhost:8983/solr/select/?defType=dismax&q=video&fl=*,score
```

You can also override which fields are searched on and how much boost each field gets.

```
http://localhost:8983/solr/select/?defType=dismax&q=video&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/?defType=dismax&q=video&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/?defType=dismax&q=video&fl=name,score,inStock
```

```
http://localhost:8983/solr/select/?defType=dismax&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, one and two word queries require that all of the optional clauses match, but for three to five word queries one missing word is allowed.

```
http://localhost:8983/solr/select/?defType=dismax&q=belkin+ipod
```

```
http://localhost:8983/solr/select/?defType=dismax&q=belkin+ipod+gibberish
```

```
http://localhost:8983/solr/select/?defType=dismax&q=belkin+ipod+apple
```

Just like the StandardRequestHandler, it supports the debugQuery option to viewing the parsed query, and the score explanations for each document.

```
http://localhost:8983/solr/select/?defType=dismax&q=belkin+ipod+gibberish&debugQuery=true
```

```
http://localhost:8983/solr/select/?defType=dismax&q=video+card&debugQuery=true
```

## The Extended DisMax Query Parser

The Extended DisMax (eDisMax) query parser is an improved version of the DisMax query parser. In addition to supporting all the DisMax query parser parameters, Extended Dismax:

- supports the full Lucene query parser syntax.
- supports queries such as AND, OR, NOT, -, and +.
- treats "and" and "or" as "AND" and "OR" in Lucene syntax mode.
- respects the 'magic field' names `_val_` and `_query_`. These are not a real fields in `schema.xml`, but if used it helps do special things (like a function query in the case of `_val_` or a nested query in the case of `_query_`). If `_val_` is used in a term or phrase query, the value is parsed as a function.
- includes improved smart partial escaping in the case of syntax errors; fielded queries, +/-, and phrase queries are still supported in this mode.
- improves proximity boosting by using word shingles; you do not need the query to match all words in the document before proximity boosting is applied.
- includes advanced stopword handling: stopwords are not required in the mandatory part of the query but are still used in the proximity boosting part. If a query consists of all stopwords, such as "to be or not to be", then all words are required.
- includes improved boost function: in Extended DisMax, the `boost` function is a multiplier rather than an addend, improving your boost results; the additive boost functions of DisMax (`bf` and `bq`) are also supported.
- supports pure negative nested queries: queries such as `+foo (-foo)` will match all documents.
- lets you specify which fields the end user is allowed to query, and to disallow direct fielded searches.

### Extended DisMax Parameters

In addition to all the DisMax parameters, Extended DisMax includes these query parameters:

#### *The `boost` Parameter*

A multivalued list of strings parsed as queries with scores multiplied by the score from the main query for all matching documents. This parameter is shorthand for wrapping the query produced by eDisMax using the `BoostQParserPlugin`

#### *The `lowercaseOperators` Parameter*

A Boolean parameter indicating if lowercase "and" and "or" should be treated the same as operators "AND" and "OR".

#### *The `ps` Parameter*

Default amount of slop on phrase queries built with `pf`, `pf2` and/or `pf3` fields (affects boosting).

#### *The `pf2` Parameter*

A multivalued list of fields with optional weights, based on pairs of word shingles.

#### *The `ps2` Parameter*

Default amount of slop on phrase queries built with `pf`, `pf2` and/or `pf3` fields (affects boosting). New with Solr 4, it is similar to `ps` but sets default slop factor for `pf2`. If not specified, `ps` is used.

#### *The `pf3` Parameter*

A multivalued list of fields with optional weights, based on triplets of word shingles. Similar to `pf`, except that instead of building a phrase per field out of all the words in the input, it builds a set of phrases for each field out of each triplet of word shingles.

#### *The `ps3` Parameter*

New with Solr 4. As with `ps` but sets default slop factor for `pf3`. If not specified, `ps` will be used.

#### *The `stopwords` Parameter*

A Boolean parameter indicating if the `StopFilterFactory` configured in the query analyzer should be respected when parsing the query: if it is false, then the `StopFilterFactory` in the query analyzer is ignored.

#### *The `uf` Parameter*

Specifies which schema fields the end user is allowed to explicitly query. This parameter supports wildcards. The default is to allow all fields, equivalent to `uf=*`. To allow only title field, use `uf=title`. To allow title and all fields ending with _s, use `uf=title,*_s`. To allow all fields except title, use `uf=*-title`. To disallow all fielded searches, use `uf=-*`.

#### *Field aliasing using per-field `qf` overrides*

Per-field overrides of the `qf` parameter may be specified to provide 1-to-many aliasing from field names specified in the query string, to field names used in the underlying query. By default, no aliasing is used and field names specified in the query string are treated as literal field names in the index.

### Examples of Queries Submitted to the Extended DisMax Query Parser

Boost the result of the query term "hello" based on the document's popularity:

```
http://localhost:8983/solr/select/?defType=edismax&q=hello&pf=text&qf=text&boost=popul
arity
```

Search for iPods OR video:

```
http://localhost:8983/solr/select/?defType=edismax&q=ipod OR video
```

Search across multiple fields, specifying (via boosts) how important each field is relative each other:

```
http://localhost:8983/solr/select/?q=video&defType=edismax&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&defType=edismax&qf=features^20.0+text^0.3&b
q=cat:electronics^5.0
```

Using the "mm" param, 1 and 2 word queries require that all of the optional clauses match, but for queries with three or more clauses one missing clause is allowed:

```
http://localhost:8983/solr/select/?q=belkin+ipod&defType=edismax&mm=2
http://localhost:8983/solr/select/?q=belkin+ipod+gibberish&defType=edismax&mm=2
http://localhost:8983/solr/select/?q=belkin+ipod+apple&defType=edismax&mm=2
```

In the example below, we see a per-field override of the `qf` parameter being used to alias "name" in the query string to either the "`last_name`" and "`first_name`" fields:

```
defType=edismax
q=sysadmin name:Mike
qf=title text last_name first_name
f.name.qf=last_name first_name
```

#### Using negative boost

Negative query boosts have been supported at the "Query" object level for a long time (resulting in negative scores for matching documents). Now the QueryParsers have been updated to handle this too.

#### Using 'slop'

`Dismax` and `Edismax` can run queries against all query fields, and also run a query in the form of a phrase against the phrase fields. (This will work only for boosting documents, not actually for matching.) However, that phrase query can have a 'slop,' which is the distance between the terms of the query while still considering it a phrase match. For example:

```
q=foo bar
qf=field1^5 field2^10
pf=field1^50 field2^20
defType=dismax
```

With these parameters, the Dismax Query Parser generates a query that looks something like this:

```
(+(field1:foo^5 OR field2:bar^10) AND (field1:bar^5 OR field2:bar^10))
```

But it also generates another query that will only be used for boosting results:

```
field1:"foo bar"^50 OR field2:"foo bar"^20
```

Thus, any document that has the terms "foo" and "bar" will match; however if some of those documents have both of the terms as a phrase, it will score much higher because it's more relevant.

If you add the parameter `ps` (phrase slop), the second query will instead be:

```
ps=10 field1:"foo bar"~10^50 OR field2:"foo bar"~10^20
```

This means that if the terms "foo" and "bar" appear in the document with less than 10 terms between each other, the phrase will match. For example the doc that says:

```
*Foo* term1 term2 term3 *bar*
```

will match the phrase query.

How does one use phrase slop? Usually it is configured in the request handler (in `solrconfig`).

With query slop (`qs`) the concept is similar, but it applies to explicit phrase queries from the user. For example, if you want to search for a name, you could enter:

```
q="Hans Anderson"
```

A document that contains "Hans Anderson" will match, but a document that contains the middle name "Christian" or where the name is written with the last name first ("Anderson, Hans") won't. For those cases one could configure the query field `qs`, so that even if the user searches for an explicit phrase query, a slop is applied.

Finally, `edismax` contains not only a phrase fields (`pf`) parameters, but also phrase and query fields 2 and 3. You can use those fields for setting different fields or boosts. Each of those can use a different phrase slop.

#### Using the 'magic fields' _val_ and _query_

If the 'magic field' name `_val_` is used in a term or phrase query, the value is parsed as a function.

The Solr Query Parser's use of `_val_` and `_query_` differs from the Lucene Query Parser in the following ways:

- If the magic field name `_val_` is used in a term or phrase query, the value is parsed as a function.
- It provides a hook into `FunctionQuery` syntax. Quotes are necessary to encapsulate the function when it includes parentheses. For example:

  ```
  _val_:myfield
  _val_:"recip(rord(myfield),1,2,3)"
  ```

- The Solr Query Parser offers nested query support for any type of query parser (via QParserPlugin). Quotes are often necessary to encapsulate the nested query if it contains reserved characters. For example:

  ```
  _query_:"{!dismax qf=myfield}how now brown cow"
  ```

Although not technically a syntax difference, note that if you use the Solr `TrieDateField` type (or the deprecated `DateField` type), any queries on those fields (typically range queries) should use either the Complete ISO 8601 Date syntax that field supports, or the DateMath Syntax to get relative dates. For example:

```
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]
```

⚠ TO must be uppercase, or Solr will report a 'Range Group' error.

## Function Queries

Function queries enable you to generate a relevancy score using the actual value of one or more numeric fields. Function queries are supported by the DisMax, Extended DisMax, and standard query parsers.

Function queries use *functions*. The functions can be a constant (numeric or string literal), a field, another function or a parameter substitution argument. You can use these functions to modify the ranking of results for users. These could be used to change the ranking of results based on a user's location, or some other calculation.

Function query topics covered in this section:

### Using Function Query

Functions must be expressed as function calls (for example, `sum(a,b)` instead of simply `a+b`).

There are several ways of using function queries in a Solr query:

- Via an explicit QParser that expects function arguments, such `func` or `frange`. For example:

```
q={!func}div(popularity,price)&fq={!frange l=1000}customer_ratings
```

- In a Sort expression. For example:

```
sort=div(popularity,price) desc, score desc
```

- Add the results of functions as psuedo-fields to documents in query results. For instance, for:

```
&fl=sum(x, y),id,a,b,c,score
```

the output would be:

```
...
<str name="id">foo</str>
<float name="sum(x,y)">40</float>
<float name="score">0.343</float>
...
```

- Use in a parameter that is explicitly for specifying functions, such as the EDisMax query parser's `boost` param, or 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"
```

- Introduce a function query inline in the lucene QParser with the `_val_` keyword. For example:

```
q=_val_:mynumericfield _val_:"recip(rord(myfield),1,2,3)"
```

Only functions with fast random access are recommended.

### Available Functions

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)`<br>`abs(-5)` |
| and | Returns a value of true if and only if all of its operands evaluate to true. | `and(not(exists(popularity)),exists(price))`: returns `true` for any document which has a value in the `price` field, but does not have a value in the `popularity` field |
| "constant" | Specifies a floating point constant. | `1.5` |
| def | `def` is short for default. Returns the value of field "field", or if the field does not exist, returns the default value specified. and yields the first value where `exists()==true`.) | `def(rating,5)`: This `def()` function returns the rating, or if no rating specified in the doc, returns 5<br>`def(myfield, 1.0)`: equivalent to `if(exists(myfield),myfield,1.0)` |
| div | Divides one value or function by another. div(x,y) divides x by y. | `div(1,y)`<br>`div(sum(x,100),max(y,1))` |

| | | |
|---|---|---|
| dist | Return the distance between two vectors (points) in an n-dimensional space. Takes in the power, plus two or more ValueSource instances and calculates the distances between the two vectors. Each ValueSource must be a number. There must be an even number of ValueSource instances passed in and the method assumes that the first half represent the first vector and the second half represent the second vector. | `dist(2, x, y, 0, 0)`: calculates the Euclidean distance between (0,0) and (x,y) for each document<br>`dist(1, x, y, 0, 0)`: calculates the Manhattan (taxicab) distance between (0,0) and (x,y) for each document<br>`dist(2, x,y,z,0,0,0):` Euclidean distance between (0,0,0) and (x,y,z) for each document.<br>`dist(1,x,y,z,e,f,g)`: Euclidean distance between (x,y,z) and (e,f,g) where each letter is a field name |
| docfreq(field,val) | Returns the number of documents that contain the term in the field. This is a constant (the same value for all documents in the index).<br>You can quote the term if it's more complex, or do parameter substitution for the term value. | `docfreq(text,'solr')`<br>`...&defType=func`<br>`&q=docfreq(text,$myterm)`<br>`&myterm=solr` |
| exists | Returns TRUE if any member of the field exists. | `exists(author)` returns TRUE for any document has a value in the "author" field.<br>`exists(query(price:5.00))` returns TRUE if "price" matches "5.00". |
| field | Returns the numeric field value of an indexed (not multi-valued) field with a maximum of one value per document. The `field()` function can be called using the name of the field as a string, or for most conventional field names simply use the field name by itself.<br>0 is returned for documents without a value in the field. | `myFloatFieldName`<br>`field("my complex float fieldName")` |

| | | |
|---|---|---|
| hsin | The Haversine distance calculates the distance between two points on a sphere when traveling along the sphere. The values must be in radians. `hsin` also take a Boolean argument to specify whether the function should convert its output to radians. | `hsin(2, true, x, y, 0, 0)` |
| idf | Inverse document frequency; a measure of whether the term is common or rare across all documents. Obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient. See also `tf`. | `idf(fieldName,'solr')`: measures the inverse of the frequency of the occurrence of the term `'solr'` in `fieldName`. |
| if | Enables conditional function queries.  In `if(test,value1,value2)`:<br>- `test` is or refers to a logical value or expression that returns a logical value (TRUE or FALSE).<br>- `value1` is the value that is returned by the function if `test` yields TRUE.<br>- `value2` is the value that is returned by the function if `test` yields FALSE.<br>An expression can be any function which outputs boolean values, or even functions returning numeric values, in which case value 0 will be interpreted as false, or strings, in which case empty string is interpreted as false. | `if(termfreq(cat,'electronics'),popularity,42)`:<br>This function checks each document for the to see if it contains the term "`electronics`" in the `cat` field. If it does, then the value of the `popularity` field is returned, otherwise the value of `42` is returned. |

| | | |
|---|---|---|
| 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)`<br>`linear(x,2,4)` returns `2*x+4` |
| log | Returns the log base 10 of the specified function. | `log(x)`<br>`log(sum(x,100))` |
| map | Maps any values of an input function x that fall within min and max inclusive to the specified target. The arguments min and max must be constants. The arguments `target` and `default` can be constants or functions. If the value of x does not fall between min and max, then either the value of x is returned, or a default value is returned if specified as a 5th argument. | `map(x,min,max,target)`<br>`map(x,0,0,1)` - changes any values of 0 to 1. This can be useful in handling default 0 values.<br>`map(x,min,max,target,default)`<br>`map(x,0,100,1,-1)` - changes any values between `0` and `100` to `1`, and all other values to `-1`.<br>`map(x,0,100,sum(x,599),docfreq(text,solr))` - changes any values between `0` and `100` to x+599, and all other values to frequency of the term 'solr' in the field text. |
| 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)` |
| maxdoc | Returns the number of documents in the index, including those that are marked as deleted but have not yet been purged. This is a constant (the same value for all documents in the index). | `maxdoc()` |

| | | |
|---|---|---|
| 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 the name of an indexed `TrieDateField`, or date math based on a constant date or `NOW`.<br>`ms()`: Equivalent to `ms(NOW)`, number of milliseconds since the epoch.<br>`ms(a):` Returns the number of milliseconds since the epoch that the argument represents.<br>`ms(a,b)` : Returns the number of milliseconds that b occurs before a (that is, a - b) | `ms(NOW/DAY)`<br>`ms(2000-01-01T00:00:00Z)`<br>`ms(mydatefield)`<br>`ms(NOW,mydatefield)`<br>`ms(mydatefield,2000-01-01T00:00:00Z)`<br>`ms(datefield1,datefield2)` |
| norm(*field*) | Returns the "norm" stored in the index for the specified field.  This is the product of the index time boost and the length normalization factor, according to the Similarity for the field. | `norm(fieldName)` |
| not | The logically negated value of the wrapped function. | `not(exists(author))`: TRUE only when `exists(author)` is false. |
| numdocs | Returns the number of documents in the index, not including those that are marked as deleted but have not yet been purged. This is a constant (the same value for all documents in the index). | `numdocs()` |
| or | A logical disjunction. | `or(value1,value2):` TRUE if either `value1` or `value2` is true. |

| | | |
|---|---|---|
| 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.<br>⚠ `ord()` depends on the position in an index and can change when other documents are inserted or deleted.<br>See also `rord` below. | `ord(myIndexedField)`<br>Example: If there were only three values ("apple","banana","pear") for a particular field X, then: `ord(X)` would be 1 for documents containing "apple", 2 for documnts containing "banana", etc... |
| pow | Raises the specified base to the specified power. `pow(x,y)` raises x to the power of y. | `pow(x,y)`<br>`pow(x,log(y))`<br>`pow(x,0.5):` the same as `sqrt` |
| product | Returns the product of multiple values or functions, which are specified in a comma-separated list. `mul(...)` may also be used as an alias for this function. | `product(x,y,...)`<br>`product(x,2)`<br>`product(x,y)`<br>`mul(x,y)` |

| | | |
|---|---|---|
| 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 de-referencing `$otherparam` or direct specification of the query string in the Local Parameters through the `v` key. | `query(subquery, default)`<br>`q=product(popularity, query({!dismax v='solr rocks'})`: returns the product of the popularity and the score of the DisMax query.<br>`q=product(popularity, query($qq))&qq={!dismax}solr rocks`: equivalent to the previous query, using parameter de-referencing.<br>`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)` where `m,a,b` are constants, and `x` is any arbitrarily complex function.<br>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)`<br>`recip(rord(creationDate),1,1000,1000)` |
| rord | Returns the reverse ordering of that returned by `ord`. | `rord(myDateField)` |

| | | |
|---|---|---|
| scale | Scales values of the function `x` such that they fall between the specified `minTarget` and `maxTarget` inclusive. The current implementation traverses all of the function values to obtain the min and max, so it can pick the correct scale.<br>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)`<br>`scale(x,1,2)`: scales the values of x such that all values will be between 1 and 2 inclusive. |
| sqedist | The Square Euclidean distance calculates the 2-norm (Euclidean distance) but does not take the square root, thus saving a fairly expensive operation. It is often the case that applications that care about Euclidean distance do not need the actual distance, but instead can use the square of the distance. There must be an even number of ValueSource instances passed in and the method assumes that the first half represent the first vector and the second half represent the second vector. | `sqedist(x_td, y_td, 0, 0)` |
| sqrt | Returns the square root of the specified value or function. | `sqrt(x)sqrt(100)sqrt(sum(x,100))` |

| | | |
|---|---|---|
| strdist | Calculate the distance between two strings. Uses the Lucene spell checker `StringDistance` interface and supports all of the implementations available in that package, plus allows applications to plug in their own via Solr's resource loading capabilities. `strdist` takes (string1, string2, distance measure). Possible values for distance measure are:<br>jw: Jaro-Winkler<br>edit: Levenstein or Edit distance<br>ngram: The NGramDistance, if specified, can optionally pass in the ngram size too. Default is 2.<br>FQN: Fully Qualified class Name for an implementation of the StringDistance interface. Must have a no-arg constructor. | `strdist("SOLR",id,edit)` |
| sub | Returns x-y from sub(x,y). | `sub(myfield,myfield2)`<br>`sub(100,sqrt(myfield))` |
| sum | Returns the sum of multiple values or functions, which are specified in a comma-separated list. `add(...)` may be used as an alias for this function | `sum(x,y,...) sum(x,1)`<br>`sum(x,y)`<br>`sum(sqrt(x),log(y),z,0.5)`<br>`add(x,y)` |

| | | |
|---|---|---|
| sumtotaltermfreq | Returns the sum of `totaltermfreq` values for all terms in the field in the entire index (i.e., the number of indexed tokens for that field). (Aliases `sumtotaltermfreq` to `sttf`.) | If doc1:(fieldX:A B C) and doc2:(fieldX:A A A A):<br>`docFreq(fieldX:A)` = 2 (A appears in 2 docs)<br>`freq(doc1, fieldX:A)` = 4 (A appears 4 times in doc 2)<br>`totalTermFreq(fieldX:A)` = 5 (A appears 5 times across all docs)<br>`sumTotalTermFreq(fieldX)` = 7 in `fieldX`, there are 5 As, 1 B, 1 C |
| termfreq | Returns the number of times the term appears in the field for that document. | `termfreq(text,'memory')` |
| tf | Term frequency; returns the term frequency factor for the given term, using the Similarity for the field. The `tf-idf` value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the document, which helps to control for the fact that some words are generally more common than others. See also `idf`. | `tf(text,'solr')` |
| 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.<br>The `ord()` and `rord()` functions implicitly use `top()`, and hence `ord(foo)` is equivalent to `top(ord(foo))`. | |
| totaltermfreq | Returns the number of times the term appears in the field in the entire index. (Aliases `totaltermfreq` to `ttf`.) | `ttf(text,'memory')` |

| | | |
|---|---|---|
| xor() | Logical exclusive disjunction, or one or the other but not both. | `xor(field1,field2)` returns TRUE if either `field1` or `field2` is true; FALSE if both are true. |

### Example Function Queries

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(x,y,z)"
```

This query will rank the results based on volumes. In order to get the computed volume, you will need to request the `score`, which will contain the resultant volume:

```
&fl=*, score
```

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(x,y,z))"&fl=boxname x y z weight score
```

### Sort By Function

You can sort your query results by the output of a function. For example, to sort results by distance, you could enter:

```
http://localhost:8983/solr/select?q=*:*&sort=dist(2, point1, point2) desc
```

Sort by function also supports pseudo-fields: fields can be generated dynamically and return results as though it was normal field in the index. For example,

```
&fl=id,sum(x, y),score
```

would return:

```
<str name="id">foo</str>
<float name="sum(x,y)">40</float>
<float name="score">0.343</float>
```

### Related Topics

- FunctionQuery

## 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.

#### Basic Syntax of Local Parameters

To specify a local parameter, insert the following before the argument to be modified:

- Begin 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.

#### 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
```

#### 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'}
```

#### 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
```

### Other Parsers

In addition to the main query parsers discussed earlier, there are several other query parsers that can be used instead of or in conjunction with the main parsers for specific purposes. This section details the other parsers, and gives examples for how they might be used.

Many of these parsers are expressed the same way as Local Parameters in Queries.

Query parsers discussed in this section:

- Block Join Query Parsers
- Boost Query Parser
- Collapsing Query Parser
- Complex Phrase Query Parser
- Field Query Parser
- Function Query Parser
- Function Range Query Parser
- Join Query Parser
- Lucene Query Parser
- Max Score Query Parser
- Nested Query Parser
- Old Lucene Query Parser
- Prefix Query Parser
- Raw Query Parser
- Re-Ranking Query Parser
- Simple Query Parser
- Spatial Filter Query Parser
- Surround Query Parser
- Switch Query Parser
- Term Query Parser
- Terms Query Parser

### Block Join Query Parsers

There are two query parsers that support block joins. These parsers allow indexing and searching for relational content that has been indexed as nested documents.

The example usage of the query parsers below assumes these two documents and each of their child documents have been indexed:

```
<add>
  <doc>
  <field name="id">1</field>
  <field name="title">Solr adds block join support</field>
  <field name="content_type">parentDocument</field>
    <doc>
     <field name="id">2</field>
     <field name="comments">SolrCloud supports it too!</field>
    </doc>
  </doc>
  <doc>
    <field name="id">3</field>
    <field name="title">Lucene and Solr 4.5 is out</field>
    <field name="content_type">parentDocument</field>
    <doc>
     <field name="id">4</field>
     <field name="comments">Lots of new features</field>
    </doc>
  </doc>
</add>
```

#### *Block Join Children Query Parser*

This parser takes a query that matches some parent documents and returns their children. The syntax for this parser is: `q={!child of=<allParents>}<someParents>`. The parameter `allParents` is a filter that matches only parent documents; here you would define the field and value that you used to identify a document as a parent. The parameter `someParents` identifies a query that will match some or all of the parent documents. The output is the children.

Using the example documents above, we can construct a query such as `q={!child of="content_type:parentDocument"}title:lucene`. We only get one document in response:

```
<result name="response" numFound="1" start="0">
   <doc>
      <str name="id">4</str>
      <str name="comments">Lots of new features</str>
   </doc>
</result>
```

#### *Block Join Parent Query Parser*

This parser takes a query that matches child documents and returns their parents. The syntax for this parser is similar: `q={!parent which=<allParents>}<someChildren>`. Again the parameter The parameter `allParents` is a filter that matches only parent documents; here you would define the field and value that you used to identify a document as a parent. The parameter `someChildren` is a query that matches some or all of the child documents. Note that the query for `someChildren` should match only child documents or you may get an exception.

Again using the example documents above, we can construct a query such as `q={!parent which="content_type:parentDocument"}comments:SolrCloud`. We get this document in response:

```
<result name="response" numFound="1" start="0">
   <doc>
      <str name="id">1</str>
      <arr name="title"><str>Solr adds block join support</str></arr>
      <arr name="content_type"><str>parentDocument</str></arr>
   </doc>
</result>
```

### Boost Query Parser

`BoostQParser` extends the `QParserPlugin` and creates a boosted query from the input value. The main value is the query to be boosted. Parameter b is the function query to use as the boost. The query to be boosted may be of any type.

Examples:

Creates a query "foo" which is boosted (scores are multiplied) by the function query `log(popularity)`:

```
{!boost b=log(popularity)}foo
```

Creates a query "foo" which is boosted by the date boosting function referenced in `ReciprocalFloatFunction`:

```
{!boost b=recip(ms(NOW,mydatefield),3.16e-11,1,1)}foo
```

### Collapsing Query Parser

The `CollapsingQParser` is really a *post filter* that provides more performant field collapsing than Solr's standard approach when the number of distinct groups in the result set is high. This parser collapses the result set to a single document per group before it forwards the result set to the rest of the search components. So all downstream components (faceting, highlighting, etc...) will work with the collapsed result set.

Details about using the `CollapsingQParser` can be found in the Collapse and Expand Results section.

### Complex Phrase Query Parser

The `ComplexPhraseQParser` provides support for wildcards, ORs, etc., inside phrase queries using Lucene's `ComplexPhraseQueryParser` . Under the covers, this query parser makes use of the Span group of queries, e.g., spanNear, spanOr, etc., and is subject to the same limitations as that family or parsers.

| Parameter | Description |
|---|---|
| `inOrder` | Set to true to force phrase queries to match terms in the order specified. Default: **true** |
| `df` | The default search field. |

**Examples:**

```
{!complexphrase inOrder=true}name:"Jo* Smith"
```

```
{!complexphrase inOrder=false}name:"(john jon jonathan~) peters*"
```

A mix of ordered and unordered complex phrase queries:

```
+_query_:"{!complexphrase inOrder=true}manu:\"a* c*\"" +_query_:"{!complexphrase
inOrder=false df=name}\"bla* pla*\""
```

#### ***Limitations***

Performance is sensitive to the number of unique terms that are associated with a pattern. For instance, searching for "a*" will form a large OR clause (technically a SpanOr with many terms) for all of the terms in your index for the indicated field that start with the single letter 'a'. It may be prudent to restrict wildcards to at least two or preferably three letters as a prefix. Allowing very short prefixes may result in to many low-quality documents being returned.

##### MaxBooleanClauses

You may need to increase MaxBooleanClauses in `solrconfig.xml` as a result of the term expansion above:

```
<maxBooleanClauses>4096</maxBooleanClauses>
```

This property is described in more detail in the section Query Sizing and Warming.

##### Stopwords

It is recommended not to use stopword elimination with this query parser. Lets say we add **the**, **up**, **to** to `stopwords.txt` for your collection, and index a document containing the text *"Stores up to 15,000 songs, 25,00 photos, or 150 yours of video"* in a field named "features".

While the query below does not use this parser:

```
q=features:"Stores up to 15,000"
```

the document is returned. The next query that *does* use the Complex Phrase Query Parser, as in this query:

```
q=features:"sto* up to 15*"&defType=complexphrase
```

does *not* return that document because SpanNearQuery has no good way to handle stopwords in a way analogous to PhraseQuery.  If you must remove stopwords for your use case, use a custom filter factory or perhaps a customized synonyms filter that reduces given stopwords to some impossible token.

#### Field Query Parser

The `FieldQParser` extends the `QParserPlugin` and creates a field query from the input value, applying text analysis and constructing a phrase query if appropriate. The parameter `f` is the field to be queried.

Example:

```
{!field f=myfield}Foo Bar
```

This example creates a phrase query with "foo" followed by "bar" (assuming the analyzer for `myfield` is a text field with an analyzer that splits on whitespace and lowercase terms). This is generally equivalent to the Lucene query parser expression `myfield:"Foo Bar"`.

#### Function Query Parser

The `FunctionQParser` extends the `QParserPlugin` and creates a function query from the input value. This is only one way to use function queries in Solr; for another, more integrated, approach, see the section on Function Queries.

Example:

```
{!func}log(foo)
```

#### Function Range Query Parser

The `FunctionRangeQParser` extends the `QParserPlugin` and creates a range query over a function. This is also referred to as `frange`, as seen in the examples below.

Other parameters:

| Parameter | Description |
|---|---|
| l | The lower bound, optional |
| u | The upper bound, optional |
| incl | Include the lower bound: true/false, optional, default=true |
| incu | Include the upper bound: true/false, optional, default=true |

Examples:

```
{!frange l=1000 u=50000}myfield
```

```
fq={!frange l=0 u=2.2} sum(user_ranking,editor_ranking)
```

Both of these examples are restricting the results by a range of values found in a declared field or a function query. In the second example, we're doing a sum calculation, and then defining only values between 0 and 2.2 should be returned to the user.

For more information about range queries over functions, see Yonik Seeley's introductory blog post Ranges over Functions in Solr 1.4, hosted at SearchHub.org.

### Join Query Parser

`JoinQParser` extends the `QParserPlugin`. It allows normalizing relationships between documents with a join operation. This is different from in concept of a join in a relational database because no information is being truly joined. An appropriate SQL analogy would be an "inner query".

Examples:

Find all products containing the word "ipod", join them against manufacturer docs and return the list of manufacturers:

```
{!join from=manu_id_s to=id}ipod
```

Find all manufacturer docs named "belkin", join them against product docs, and filter the list to only products with a price less than $12:

```
q  = {!join from=id to=manu_id_s}compName_s:Belkin
fq = price:[* TO 12]
```

For more information about join queries, see the Solr Wiki page on Joins. Erick Erickson has also written a blog post about join performance called Solr and Joins, hosted by SearchHub.org.

### Lucene Query Parser

The `LuceneQParser` extends the `QParserPlugin` by parsing Solr's variant on the Lucene QueryParser syntax. This is effectively the same query parser that is used in Lucene. It uses the operators `q.op`, the default operator ("OR" or "AND") and `df`, the default field name.

Example:

```
{!lucene q.op=AND df=text}myfield:foo +bar -baz
```

For more information about the syntax for the Lucene Query Parser, see the Classic QueryParser javadocs.

### Max Score Query Parser

The `MaxScoreQParser` extends the `LuceneQParser` but returns the Max score from the clauses. It does this by wrapping all `SHOULD` clauses in a `DisjunctionMaxQuery` with tie=1.0. Any `MUST` or `PROHIBITED` clauses are passed through as-is. Non-boolean queries, e.g. NumericRange falls-through to the `LuceneQParser` parser behavior.

Example:

```
{!maxscore tie=0.01}C OR (D AND E)
```

### Nested Query Parser

The `NestedParser` extends the `QParserPlugin` and creates a nested query, with the ability for that query to redefine its type via local parameters. This is useful in specifying defaults in configuration and letting clients indirectly reference them.

Example:

```
{!query defType=func v=$q1}
```

If the `q1` parameter is price, then the query would be a function query on the price field. If the `q1` parameter is {!lucene}inStock:true}} then a term query is created from the Lucene syntax string that matches documents with `inStock=true`. These parameters would be defined in `solrconfig.xml`, in the `defaults` section:

```
<lst name="defaults"
   <str name="q1">{!lucene}inStock:true</str>
</lst>
```

For more information about the possibilities of nested queries, see Yonik Seeley's blog post Nested Queries in Solr, hosted by SearchHub.org.

#### Old Lucene Query Parser

`OldLuceneQParser` extends the `QParserPlugin` by parsing Solr's variant of Lucene's QueryParser syntax, including the deprecated sort specification after the query.

Example:

```
{!lucenePlusSort} myfield:foo +bar -baz;price asc
```

#### Prefix Query Parser

`PrefixQParser` extends the `QParserPlugin` by creating a prefix query from the input value. Currently no analysis or value transformation is done to create this prefix query. The parameter is `f`, the field. The string after the prefix declaration is treated as a wildcard query.

Example:

```
{!prefix f=myfield}foo
```

This would be generally equivalent to the Lucene query parser expression `myfield:foo*`.

#### Raw Query Parser

`RawQParser` extends the `QParserPlugin` by creating a term query from the input value without any text analysis or transformation. This is useful in debugging, or when raw terms are returned from the terms component (this is not the default). The only parameter is `f`, which defines the field to search.

Example:

```
{!raw f=myfield}Foo Bar
```

This example constructs the query: `TermQuery(Term("myfield","Foo Bar"))`.

For easy filter construction to drill down in faceting, the TermQParserPlugin is recommended. For full analysis on all fields, including text fields, you may want to use the FieldQParserPlugin.

### Re-Ranking Query Parser

The `ReRankQParserPlugin` is a special purpose parser for Re-Ranking the top result of a simple query using a more complex ranking query.

Details about using the `ReRankQParserPlugin` can be found in the Other Parsers section.

### Simple Query Parser

The Simple query parser in Solr is based on Lucene's SimpleQueryParser. This query parser is designed to allow users to enter queries however they want, and it will do its best to interpret the query and return results.

This parser takes the following parameters:

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>q.operator</td><td>Enables specific operations for parsing. By default, all operations are enabled, and this can be used to disable specific operations as needed. Passing an empty string with this parameter disables all operations.
<table>
<tr><th>Operator</th><th>Description</th><th>Example</th></tr>
<tr><td>+</td><td>Specifies AND</td><td><code>token1+token2</code></td></tr>
<tr><td>|</td><td>Specifies OR</td><td><code>token1|token2</code></td></tr>
<tr><td>-</td><td>Specifies NOT</td><td><code>-token3</code></td></tr>
<tr><td>"</td><td>Creates a phrase</td><td><code>"term1 term2"</code></td></tr>
<tr><td>*</td><td>Specifies a prefix query</td><td><code>term*</code></td></tr>
<tr><td>~N</td><td>At the end of terms, specifies a fuzzy query</td><td><code>term~1</code></td></tr>
<tr><td>~N</td><td>At the end of phrases, specifies a NEAR query</td><td><code>"term1 term2"~5</code></td></tr>
<tr><td>()</td><td>Specifies precedence; tokens inside the parenthesis will be analyzed first. Otherwise, normal order is left to right.</td><td><code>token1 + (token2 | token3)</code></td></tr>
</table>
If needed, operations can be escaped with the <code>/</code> character.</td></tr>
<tr><td>q.op</td><td>Defines an operator to use by default if none are defined by the user. By default, OR is defined; an alternative option is AND.</td></tr>
<tr><td>qf</td><td>A list of query fields and boosts to use when building the query.</td></tr>
<tr><td>df</td><td>Defines the default field if none is defined in <code>schema.xml</code>, or overrides the default field if it is already defined.</td></tr>
</table>

Any errors in syntax are ignored and the query parser will interpret as best it can. This can mean, however, odd results in some cases.

#### Spatial Filter Query Parser

`SpatialFilterQParser` extends the `QParserPlugin` by creating a spatial Filter based on the type of spatial point used. The field must implement SpatialQueryable. All units are in Kilometers.

This query parser takes the following parameters:

| Parameter | Description |
|---|---|
| sfield | The field on which to filter. Required. |
| pt | The point to use as a reference. Must match the dimension of the field. Required. |
| d | The distance in km. Required. |

The distance measure used currently depends on the FieldType. `LatLonType` defaults to using haversine, `PointType` defaults to Euclidean (2-norm).

This example shows the syntax:

```
{!geofilt sfield=<location_field> pt=<lat,lon> d=<distance>}
```

Here are some examples with values configured:

```
fq={!geofilt sfield=store pt=10.312,-20.556 d=3.5}
```

```
fq={!geofilt sfield=store}&pt=10.312,-20&d=3.5
```

```
fq={!geofilt}&sfield=store&pt=10.312,-20&d=3.5
```

If using `geofilt` with `LatLonType`, it is capable of producing scores equal to the computed distance from the point to the field, making it useful as a component of the main query or a boosting query.

There is more information about spatial searches available in the section Spatial Search.

#### Surround Query Parser

`SurroundQParser` extends the `QParserPlugin`. This provides support for the Surround query syntax, which provides proximity search functionality. There are two operators: `w` creates an ordered span query and `n` creates an unordered one. Both operators take a numeric value to indicate distance between two terms. The default is 1, and the maximum is 99. Note that the query string is not analyzed in any way.

Example:

```
{!surround} 3w(foo, bar)
```

This example would find documents where the terms "foo" and "bar" were no more than 3 terms away from each other (i.e., no more than 2 terms between them).

This query parser will also accept boolean operators (AND, OR, and NOT, in either upper- or lowercase), wildcards, quoting for phrase searches, and boosting. The `w` and `n` operators can also be expressed in upper- or lowercase.

More information about Surround queries can be found at http://wiki.apache.org/solr/SurroundQueryParser.

### Switch Query Parser

`SwitchQParser` is a `QParserPlugin` that acts like a "switch" or "case" statement.

The primary input string is trimmed and then prefixed with `case.` for use as a key to lookup a "switch case" in the parser's local params. If a matching local param is found the resulting param value will then be parsed as a subquery, and returned as the parse result.

The `case` local param can be optionally be specified as a switch case to match missing (or blank) input strings. The `default` local param can optionally be specified as a default case to use if the input string does not match any other switch case local params. If default is not specified, then any input which does not match a switch case local param will result in a syntax error.

In the examples below, the result of each query is "XXX":

```
{!switch case.foo=XXX case.bar=zzz case.yak=qqq}foo
```

```
{!switch case.foo=qqq case.bar=XXX case.yak=zzz} bar  // extra whitespace is trimmed
```

```
{!switch case.foo=qqq case.bar=zzz default=XXX}asdf   // fallback to the default
```

```
{!switch case=XXX case.bar=zzz case.yak=qqq}          // blank input uses 'case'
```

A practical usage of this `QParsePlugin`, is in specifying `appends` fq params in the configuration of a SearchHandler, to provide a fixed set of filter options for clients using custom parameter names. Using the example configuration below, clients can optionally specify the custom parameters `in_stock` and `shipping` to override the default filtering behavior, but are limited to the specific set of legal values (shipping=any|free, in_stock=yes|no|all).

```
<requestHandler name="/select" class="solr.SearchHandler">
   <lst name="defaults">
     <str name="in_stock">yes</str>
     <str name="shipping">any</str>
   </lst>
   <lst name="appends">
     <str name="fq">{!switch case.all='*:*'
                             case.yes='inStock:true'
                             case.no='inStock:false'
                             v=$in_stock}</str>
     <str name="fq">{!switch case.any='*:*'
                             case.free='shipping_cost:0.0'
                             v=$shipping}</str>
   </lst>
 </requestHandler>
```

### Term Query Parser

`TermQParser` extends the `QParserPlugin` by creating a single term query from the input value equivalent to `readableToIndexed()`. This is useful for generating filter queries from the external human readable terms returned by the faceting or terms components. The only parameter is `f`, for the field.

Example:

```
{!term f=weight}1.5
```

For text fields, no analysis is done since raw terms are already returned from the faceting and terms components. To apply analysis to text fields as well, see the Field Query Parser, above.

If no analysis or transformation is desired for any type of field, see the Raw Query Parser, above.

#### Terms Query Parser

`TermsQParser` functions similarly to the Term Query Parser but takes in multiple values separated by commas and returns documents matching any of the specified values.  This can be useful for generating filter queries from the external human readable terms returned by the faceting or terms components, and may be more efficient in some cases then using the Standard Query Parser to generate an boolean query since the default implementation "`method`" avoids scoring.

This query parser takes the following parameters:

| Parameter | Description |
|---|---|
| f | The field on which to search. Required. |
| separator | Separator to use when parsing the input. If set to " " (a single blank space), will trim additional white space from the input terms. Defaults to "`,`". |
| method | The internal implementation to requested for building the query: `termsFilter`, `booleanQuery`, `automaton`, or `docValuesTermsFilter`. Defaults to "`termsFilter`". |

Examples:

```
{!terms f=tags}software,apache,solr,lucene
```

```
{!terms f=categoryId method=booleanQuery separator=" "}8  6 7 5309
```

## Faceting

As described in the section Overview of Searching in Solr, 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 are looking for.

Topics covered in this section:

### General Parameters

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.

#### The `facet` Parameter

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.

#### The `facet.query` Parameter

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
```

### 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 Documents, Fields, and Schema Design.)

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 | (Advanced) 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. |
| facet.overrequest.count | (Advanced) A number of documents, beyond the effective `facet.limit` to request from each shard in a distributed search |
| facet.overrequest.ratio | (Advanced) A multiplier of the effective `facet.limit` to request from each shard in a distributed search |
| facet.threads | (Advanced) Controls parallel execution of field faceting |

These parameters are described in the sections below.

#### 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.

> ⚠ 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.

#### The `facet.prefix` Parameter

The `facet.prefix` parameter limits the terms on which to facet to those starting with the given string prefix. This does not limit the query in any way, only the facets that would be returned in response to the query.

This parameter can be specified on a per-field basis with the syntax of `f.<fieldname>.facet.prefix`.

#### The `facet.sort` Parameter

This parameter determines the ordering of the facet field constraints.

> ⚠ The true/false values for this parameter were 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 with the syntax of `f.<fieldname>.facet.sort`.

#### The `facet.limit` Parameter

This parameter specifies the maximum number of constraint counts (essentially, the number of facets for a field that are returned) 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 with the syntax of `f.<fieldname>.facet.limit`.

#### 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 with the syntax of `f.<fieldname>.facet.offset`.

#### 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 with the syntax of `f.<fieldname>.facet.mincount`.

#### 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 results that match the query but which have no facet value for the field should be computed and returned in the response.

The default value is false.

This parameter can be specified on a per-field basis with the syntax of `f.<fieldname>.facet.missing`.

#### The `facet.method` Parameter

The facet.method parameter selects the type of algorithm or method Solr should use when faceting a field.

| 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 such as `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` stand for field cache. |
| fcs | Per-segment field faceting for single-valued string fields. Enable with `facet.method=fcs` and control the number of threads used with the `threads` local parameter. This parameter allows faceting to be faster in the presence of rapid index changes. |

The default value is `fc` (except for fields using the `BoolField` field type) 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 with the syntax of `f.<fieldname>.facet.method`.

#### 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 with the syntax of `f.<fieldname>.facet.enum.cache.minDF`.

#### Over-Request Parameters

In some situations, the accuracy in selecting the "top" constraints returned for a facet in a distributed Solr query can be improved by "Over Requesting" the number of desired constraints (ie: `facet.limit`) from each of the individual Shards.  In these situations, each shard is by default asked for the top "`10 + (1.5 * facet.limit)`" constraints.

In some situations, depending on how your docs are partitioned across your shards, and what `facet.limit` value you used, you may find it advantageous to increase or decrease the amount of over-requesting Solr does.  This can be achieved by setting the `facet.overrequest.count` (defaults to 10) and `facet.overrequest.ratio` (defaults to 1.5) parameters.

#### The `facet.threads` Parameter

This param will cause loading the underlying fields used in faceting to be executed in parallel with the number of threads specified. Specify as `facet.threads=N` where `N` is the maximum number of threads used. Omitting this parameter or specifying the thread count as 0 will not spawn any threads, and only the main request thread will be used. Specifying a negative number of threads will create up to Integer.MAX_VALUE threads.

### Range Faceting

You can use Range Faceting on any date field or any numeric field that supports range queries. This is particularly useful for stitching together a series of range queries (as facet by query) for things like prices. As of Solr 3.1, Range Faceting is preferred over Date Faceting (described below).

| Parameter | Description |
|---|---|
| facet.range | Specifies the field to facet by range. |
| facet.range.start | Specifies the start of the facet range. |
| facet.range.end | Specifies the end of the facet range. |
| facet.range.gap | Specifies the span of the range as a value to be added to the lower bound. |
| facet.range.hardend | A boolean parameter that specifies how Solr handles a range gap that cannot be evenly divided between the range start and end values. If true, the last range constraint will have the `facet.range.end` value an upper bound. If false, the last range will have the smallest possible upper bound greater then `facet.range.end` such that the range is the exact width of the specified range gap. The default value for this parameter is false. |
| facet.range.include | Specifies inclusion and exclusion preferences for the upper and lower bounds of the range. See the `facet.range.include` topic for more detailed information. |

| | |
|---|---|
| facet.range.other | Specifies counts for Solr to compute in addition to the counts for each facet range constraint. |

#### The `facet.range` Parameter

The `facet.range` parameter defines the field for which Solr should create range facets. For example:

`facet.range=price&facet.range=age`

#### The `facet.range.start` Parameter

The `facet.range.start` parameter specifies the lower bound of the ranges. You can specify this parameter on a per field basis with the syntax of `f.<fieldname>.facet.range.start`. For example:

`f.price.facet.range.start=0.0&f.age.facet.range.start=10`

#### The `facet.range.end` Parameter

The facet.range.end specifies the upper bound of the ranges. You can specify this parameter on a per field basis with the syntax of `f.<fieldname>.facet.range.end`. For example:

`f.price.facet.range.end=1000.0&f.age.facet.range.start=99`

#### The `facet.range.gap` Parameter

The span of each range expressed as a value to be added to the lower bound. For date fields, this should be expressed using the `DateMathParser` syntax (such as, `facet.range.gap=%2B1DAY ... '+1DAY'`). You can specify this parameter on a per-field basis with the syntax of `f.<fieldname>.facet.range.gap`. For example:

`f.price.facet.range.gap=100&f.age.facet.range.gap=10`

Gaps can also be variable width by passing in a comma separated list of the gap size to be used. The last gap specified will be used to fill out all remaining gaps if the number of gaps given does not go evenly into the range. Variable width gaps are useful, for example, in spatial applications where one might want to facet by distance into three buckets: walking (0-5KM), driving (5-100KM), or other (100KM+). For example:

`facet.date.gap=1,2,3,10`

This creates 4+ buckets of size, 1, 2, 3 and then 0 or more buckets of 10 days each, depending on the start and end values.

#### The `facet.range.hardend` Parameter

The `facet.range.hardend` parameter is a Boolean parameter that specifies how Solr should handle cases where the `facet.range.gap` does not divide evenly between `facet.range.start` and `facet.range.end`. If **true**, the last range constraint will have the `facet.range.end` value as an upper bound. If **false**, the last range will have the smallest possible upper bound greater then `facet.range.end` such that the range is the exact width of the specified range gap. The default value for this parameter is false.

This parameter can be specified on a per field basis with the syntax `f.<fieldname>.facet.range.hardend`.

#### The `facet.range.include` Parameter

By default, the ranges used to compute range faceting between `facet.range.start` and `facet.range.end` are inclusive of their lower bounds and exclusive of the upper bounds. The "before" range defined with the `facet.rang` e.other parameter is exclusive and the "after" range is inclusive. This default, equivalent to "lower" below, will not result in double counting at the boundaries. You can use the facet.range.include parameter to modify this behavior using the following options:

| Option | Description |
|---|---|
| lower | All gap-based ranges include their lower bound. |
| upper | All gap-based ranges include their upper bound. |
| edge | The first and last gap ranges include their edge bounds (lower for the first one, upper for the last one) even if the corresponding upper/lower option is not specified. |
| outer | The "before" and "after" ranges will be inclusive of their bounds, even if the first or last ranges already include those boundaries. |
| all | Includes all options: lower, upper, edge, outer. |

You can specify this parameter on a per field basis with the syntax of f.<fieldname>.facet.range.include, and you can specify it multiple times to indicate multiple choices.

ⓘ To ensure you avoid double-counting, do not choose both lower and upper, do not choose outer, and do not choose all.

#### The facet.range.other Parameter

The facet.range.other parameter specifies that in addition to the counts for each range constraint between facet.range.start and facet.range.end, counts should also be computed for these options:

| Option | Description |
|---|---|
| before | All records with field values lower then lower bound of the first range. |
| after | All records with field values greater then the upper bound of the last range. |
| between | All records with field values between the start and end bounds of all ranges. |
| none | Do not compute any counts. |
| all | Compute counts for before, between, and after. |

This parameter can be specified on a per field basis with the syntax of f.<fieldname>.facet.range.other. In addition to the all option, this parameter can be specified multiple times to indicate multiple choices, but none will override all other options.

ⓘ **Date Ranges & Time Zones**
Range faceting on date fields is a common situation where the TZ parameter can be useful to ensure that the "facet counts per day" or "facet counts per month" are based on a meaningful definition of when a given day/month "starts" relative to a particular TimeZone.

For more information, see the examples in the Working with Dates section.

### Date Faceting Parameters

Date faceting using the type specific `facet.date` parameters has been deprecated since Solr 3.1. Existing users are encouraged to switch to using the more general Range Faceting, which provides the same features for date fields, but can als work with any numeric field.

The response format is slightly different, but the request parameters are virtually identical.

### Pivot (Decision Tree) Faceting

Pivoting is a summarization tool that lets you automatically sort, count, total or average data stored in a table. It displays the results in a second table showing the summarized data. Pivot faceting lets you create a summary table of the results from a query across numerous documents. With Solr 4, pivot faceting supports nested facet queries, not just facet fields.

Another way to look at it is that the query produces a Decision Tree, in that Solr tells you "for facet A, the constraints/counts are X/N, Y/M, etc. If you were to constrain A by X, then the constraint counts for B would be S/P, T/Q, etc.". In other words, it tells you in advance what the "next" set of facet results would be for a field if you apply a constraint from the current facet results.

#### facet.pivot

The `facet.pivot` parameter defines the fields to use for the pivot. Multiple `facet.pivot` values will create multiple "facet_pivot" sections in the response. Separate each list of fields with a comma.

#### facet.pivot.mincount

The `facet.pivot.mincount` parameter defines the minimum number of documents that need to match in order for the facet to be included in results. The default is 1.

For example, we can use Solr's example data set to make a query like this:

```
http://localhost:8983/solr/select?q=*:*&facet.pivot=cat,popularity,inStock
   &facet.pivot=popularity,cat&facet=true&facet.field=cat&facet.limit=5
   &rows=0&wt=json&indent=true&facet.pivot.mincount=2
```

This query will returns the data below, with the pivot faceting results found in the section "facet_pivot":

```
"facet_counts":{
    "facet_queries":{},
    "facet_fields":{
      "cat":[
        "electronics",14,
        "currency",4,
        "memory",3,
        "connector",2,
        "graphics card",2]},
    "facet_dates":{},
    "facet_ranges":{},
    "facet_pivot":{
      "cat,popularity,inStock":[{
          "field":"cat",
          "value":"electronics",
          "count":14,
          "pivot":[{
              "field":"popularity",
              "value":6,
              "count":5,
              "pivot":[{
                  "field":"inStock",
                  "value":true,
                  "count":5}]},
...
```

#### Additional Pivot Parameters

Although `facet.pivot.mincount` deviates in name from the `facet.mincount` parameter used by field faceting, many other Field faceting parameters described above can also be used with pivot faceting:

- `facet.limit`
- `facet.offset`
- `facet.sort`
- `facet.overrequest.count`
- `facet.overrequest.ratio`

### Interval Faceting

Another supported form of faceting is “Interval Faceting”. This sounds similar to “Range Faceting”, but the functionality is really closer to doing “Facet Queries” with range queries. Interval Faceting allows you to set variable intervals and count the number of documents that have values within those intervals in the specified field. **In order to use Interval Faceting on a field, it is required that the field has “docValues” enabled**. Even though the same functionality can be achieved by using facet query with range queries, the implementation of these two methods is very different and will provide different performance depending on the context. If you are concerned about the performance of your searches you should test with both options. Interval Faceting tends to be better with multiple intervals for the same fields, while facet query tend to be better in environments where cache is more effective (static indexes for example).

| Name | What it does |
|---|---|
| facet.interval | Specifies the field to facet by interval |
| facet.interval.set | Sets the intervals for the field |

#### The `facet.interval` parameter

This parameter Indicates the field where interval faceting must be applied. It can be used multiple times in the same request to indicate multiple fields. All the interval fields must have docValues=“true” in the schema.

```
facet.interval=price&facet.interval=size
```

#### The `facet.interval.set` parameter

This parameter is used to set the intervals for the field, it can be specified multiple times to indicate multiple intervals. This parameter is global, which means that it will be used for all fields indicated with facet.interval unless there is an override for a specific field. To override this parameter on a specific field you can use: f.<fieldname>.facet.interval.set, for example:

```
f.price.facet.interval.set=[0,10]&f.price.facet.interval.set=(10,100]
```

#### Interval Syntax

Intervals must begin with either '(' or '[', be followed by the start value, then a comma ',', the end value, and finally ')' or ']'.

For example:

- (1,10) -> will include values greater than 1 and lower than 10
- [1,10) -> will include values greater or equal to 1 and lower than 10
- [1,10] -> will include values greater or equal to 1 and lower or equal to 10

The initial and end values can't be empty, if the interval needs to be unbounded, the special character '*' can be used for both, start and end limit. When using '*', '(' and '[', and ')' and ']' will be treated equal. [*,*] will include all documents with a value in the field. The interval limits may be strings, there is no need to add quotes, all the text until the comma will be treated as the start limit, and the text after that will be the end limit, for example: [Buenos Aires,New York]. Keep in mind that a string-like comparison will be done to match documents in string intervals (case-sensitive). The comparator can't be changed. Commas, brackets and square brackets can be escaped by using '\' in front of them. Whitespaces before and after the values will be omitted. Start limit can't be grater than the end limit. Equal limits are allowed, this allows you to indicate the specific values that you want to count, like [A,A], [B,B] and [C,Z].

### Local Parameters for Faceting

The LocalParams syntax allows overriding global settings. It can also provide a method of adding metadata to other parameter values, much like XML attributes.

#### 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=true&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=true&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.

#### 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 field facet results for the "doctype" field to be returned using the key "mylabel" rather than "doctype" in the response. This can be helpful when faceting on the same field multiple times with different exclusions.

### Related Topics

- SimpleFacetParameters from the Solr Wiki.

## Highlighting

Highlighting in Solr allows fragments of documents that match the user's query to be included with the query response. The fragments are included in a special section of the response (the `highlighting` section), and the client uses the formatting clues also included to determine how to present the snippets to users.

Solr provides a collection of highlighting utilities which allow a great deal of control over the fields fragments are taken from, the size of fragments, and how they are formatted. The highlighting utilities can be called by various Request Handlers and can be used with the DisMax, Extended DisMax, or standard query parsers.

There are three highlighting implementations available:

- **Standard Highlighter**: The Standard Highlighter is the swiss-army knife of the highlighters. It has the most sophisticated and fine-grained query representation of the three highlighters. For example, this highlighter is capable of providing precise matches even for advanced queryparsers such as the `surround` parser. It does not require any special datastructures such as `termVectors`, although it will use them if they are present. If they are not, this highlighter will re-analyze the document on-the-fly to highlight it. This highlighter is a good choice for a wide variety of search use-cases.
- **FastVector Highlighter**: The FastVector Highlighter requires term vector options (`termVectors`, `termPositions`, and `termOffsets`) on the field, and is optimized with that in mind. It tends to work better for more languages than the Standard Highlighter, because it supports Unicode breakiterators. On the other hand, its query-representation is less advanced than the Standard Highlighter: for example it will not work well with the `surround` parser. This highlighter is a good choice for large documents and highlighting text in a variety of languages.

- **Postings Highlighter**: The Postings Highlighter requires `storeOffsetsWithPositions` to be configured on the field. This is a much more compact and efficient structure than term vectors, but is not appropriate for huge numbers of query terms (e.g. wildcard queries). Like the FastVector Highlighter, it supports Unicode algorithms for dividing up the document. On the other hand, it has the most coarse query-representation: it focuses on summary quality and ignores the structure of the query completely, ranking passages based solely on query terms and statistics. This highlighter a good choice for classic full-text keyword search.

### Configuring Highlighting

The configuration for highlighting, whichever implementation is chosen, is first to configure a search component and then reference the component in one or more request handlers.

The exact parameters for the search component vary depending on the implementation, but there is a robust example in the default `solrconfig.xml` that ships with Solr out of the box. This example includes examples of how to configure both the Standard Highlighter and the FastVector Highlighter (see the Postings Highlighter section for details on how to configure that implementation).

## Standard Highlighter

The standard highlighter doesn't require any special indexing parameters on the fields to highlight, however you can optionally turn on `termVectors`, `termPositions`, and `termOffsets` for each field to be highlighted. This will avoid having to run documents through the analysis chain at query-time and should make highlighting faster.

#### Standard Highlighting Parameters

The table below describes Solr's parameters for the Standard highlighter. These parameters can be defined in the highlight search component, as defaults for the specific request handler, or passed to the request handler with the query.

| Parameter | Default Value | Description |
|---|---|---|
| hl | blank (no highlight) | 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. |
| hl.q | blank | Specifies an overriding query term for highlighting. If `hl.q` is specified, the highlighter will use that term rather than the main query term. |
| hl.qparser | blank | Specifies a qparser to use for the hl.q query. If blank, will use the defType of the overall query. |

| | | |
|---|---|---|
| hl.fl | blank | 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.<br>A '*' can be used to match field globs, such as 'text_*' or even '*' to highlight on all fields where highlighting is possible. When using '*', consider adding `hl.requireFieldMatch=true`. |
| hl.snippets | 1 | Specifies maximum number of highlighted snippets to generate per field. It is possible for any number of snippets from zero to this value to be generated. This parameter accepts per-field overrides. |
| hl.fragsize | 100 | Specifies the size, in characters, of fragments to consider for highlighting. **0** indicates that no fragmenting should be considered and the whole field value should be used. This parameter accepts per-field overrides. |
| hl.mergeContiguous | false | Instructs Solr to collapse contiguous fragments into a single fragment. A value of **true** indicates contiguous fragments will be collapsed into single fragment. This parameter accepts per-field overrides. The default value, **false**, is also the backward-compatible setting. |
| hl.requireFieldMatch | false | If set to **true**, highlights terms only if they appear in the specified field. If **false**, terms are highlighted in all requested fields regardless of which field matched the query. |
| hl.maxAnalyzedChars | 51200 | Specifies the number of characters into a document that Solr should look for suitable snippets. |
| hl.maxMultiValuedToExamine | `integer.MAX_VALUE` | Specifies the maximum number of entries in a multi-valued field to examine before stopping. This can potentially return zero results if the limit is reached before any matches are found. If used with the `maxMultiValuedToMatch`, whichever limit is reached first will determine when to stop looking. |
| hl.maxMultiValuedToMatch | `integer.MAX_VALUE` | Specifies the maximum number of matches in a multi-valued field that are found before stopping. If `hl.maxMultiValuedToExamine` is also defined, whichever limit is reached first will determine when to stop looking. |

| | | |
|---|---|---|
| hl.alternateField | blank | Specifies a field to be used as a backup default summary if Solr cannot generate a snippet (i.e., because no terms match). This parameter accepts per-field overrides. |
| hl.maxAlternateFieldLength | unlimited | 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. This parameter is only used in conjunction with the `hl.alternateField` parameter. |
| hl.formatter | simple | 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. |
| hl.simple.pre<br>hl.simple.post | <em> and </em> | Specifies the text that should appear before (`hl.simple.pre`) and after (`hl.simple.post`) a highlighted term, when using the simple formatter. This parameter accepts per-field overrides. |
| hl.fragmenter | gap | Specifies a text snippet generator for highlighted text. The standard fragmenter is **gap**, which 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. This parameter accepts per-field overrides. |
| hl.usePhraseHighlighter | true | If set to **true**, Solr will use the Lucene SpanScorer class to highlight phrase terms only when they appear within the query phrase in the document. |
| hl.highlightMultiTerm | true | If set to **true**, Solr will use highlight phrase terms that appear in multi-term queries. |
| hl.regex.slop | 0.6 | When using the regex fragmenter (`hl.fragmenter=regex`), this parameter defines the factor by which the 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 `hl.fragsize=100` should yield fragments between 80 and 120 characters in length.<br>It is usually good to provide a slightly smaller `hl.fragsize` value when using the regex fragmenter. |
| hl.regex.pattern | blank | Specifies the regular expression for fragmenting. This could be used to extract sentences. |

| | | |
|---|---|---|
| hl.regex.maxAnalyzedChars | 10000 | 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. |
| hl.preserveMulti | false | If **true**, multi-valued fields will return all values in the order they were saved in the index. If **false**, only values that match the highlight request will be returned. |

#### Related Content

- HighlightingParameters from the Solr wiki
- Highlighting javadocs

## FastVector Highlighter

The FastVectorHighlighter is a TermVector-based highlighter that offers higher performance than the standard highlighter in many cases. To use the FastVectorHighlighter, set the `hl.useFastVectorHighlighter` parameter to `true`.

You must also turn on `termVectors`, `termPositions`, and `termOffsets` for each field that will be highlighted. Lastly, you should use a boundary scanner to prevent the FastVectorHighlighter from truncating your terms. In most cases, using the `breakIterator` boundary scanner will give you excellent results. See the section Using Boundary Scanners with the Fast Vector Highlighter for more details about boundary scanners.

#### FastVector Highlighter Parameters

The table below describes Solr's parameters for this highlighter, many of which overlap with the standard highlighter. These parameters can be defined in the highlight search component, as defaults for the specific request handler, or passed to the request handler with the query.

| Parameter | Default | Description |
|---|---|---|
| hl | blank (no highlighting) | When set to **true**, enables highlighted snippets to be generated in the query response. A **false** or blank value disables highlighting. |
| hl.useFastVectorHighligter | false | When set to **true**, enables the FastVector Highlighter. |
| hl.q | blank | Specifies an overriding query term for highlighting. If `hl.q` is specified, the highlighter will use that term rather than the main query term. |

| | | |
|---|---|---|
| hl.fl | blank | 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.<br>A '*' can be used to match field globs, such as 'text_*' or even '*' to highlight on all fields where highlighting is possible. When using '*', consider adding `hl.requireFieldMatch=true`. |
| hl.snippets | 1 | Specifies maximum number of highlighted snippets to generate per field. It is possible for any number of snippets from zero to this value to be generated. This parameter accepts per-field overrides. |
| hl.fragsize | 100 | Specifies the size, in characters, of fragments to consider for highlighting. **0** indicates that no fragmenting should be considered and the whole field value should be used. This parameter accepts per-field overrides. |
| hl.requireFieldMatch | false | If set to **true**, highlights terms only if they appear in the specified field. If **false**, terms are highlighted in all requested fields regardless of which field matched the query. |
| hl.maxMultiValuedToExamine | `integer.MAX_VALUE` | Specifies the maximum number of entries in a multi-valued field to examine before stopping. This can potentially return zero results if the limit is reached before any matches are found. If used with the `maxMultiValuedToMatch`, whichever limit is reached first will determine when to stop looking. |
| hl.maxMultiValuedToMatch | `integer.MAX_VALUE` | Specifies the maximum number of matches in a multi-valued field that are found before stopping. If `hl.maxMultiValuedToExamine` is also defined, whichever limit is reached first will determine when to stop looking. |
| hl.alternateField | blank | Specifies a field to be used as a backup default summary if Solr cannot generate a snippet (i.e., because no terms match). This parameter accepts per-field overrides. |

| | | |
|---|---|---|
| hl.maxAlternateFieldLength | unlimited | 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. This parameter is only used in conjunction with the `hl.alternateField` parameter. |
| hl.tag.pre<br>hl.tag.post | <em> and </em> | Specifies the text that should appear before (`hl.tag.pre`) and after (`hl.tag.post`) a highlighted term. This parameter accepts per-field overrides. |
| hl.phraseLimit | `integer.MAX_VALUE` | To improve the performance of the FastVectorHighlighter, you can set a limit on the number (int) of phrases to be analyzed for highlighting. |
| hl.usePhraseHighlighter | true | If set to **true**, Solr will use the Lucene SpanScorer class to highlight phrase terms only when they appear within the query phrase in the document. |
| hl.preserveMulti | false | If **true**, multi-valued fields will return all values in the order they were saved in the index. If **false**, the default, only values that match the highlight request will be returned. |
| hl.fragListBuilder | weighted | The snippet fragmenting algorithm. The **weighted** fragListBuilder uses IDF-weights to order fragments. Other options are **single**, which returns the entire field contents as one snippet, or **simple**. You can select a fragListBuilder with this parameter, or modify an existing implementation in `solrconfig.xml` to be the default by adding "default=true". |
| hl.fragmentsBuilder | default | The fragments builder is responsible for formatting the fragments, which uses <em> and </em> markup (if `hl.tag.pre` and `hl.tag.post` are not defined). Another pre-configured choice is **colored**, which is an example of how to use the fragments builder to insert HTML into the snippets for colored highlights if you choose. You can also implement your own if you'd like. You can select a fragments builder with this parameter, or modify an existing implementation in `solrconfig.xml` to be the default by adding "default=true". |

### Using Boundary Scanners with the Fast Vector Highlighter

The Fast Vector Highlighter will occasionally truncate highlighted words. To prevent this, implement a boundary scanner in `solrconfig.xml`, then use the `hl.boundaryScanner` parameter to specify the boundary scanner for highlighting.

Solr supports two boundary scanners: `breakIterator` and `simple`.

#### *The `breakIterator` Boundary Scanner*

The `breakIterator` boundary scanner offers excellent performance right out of the box by taking locale and boundary type into account. In most cases you will want to use the `breakIterator` boundary scanner. To implement the `breakIterator` boundary scanner, add this code to the `highlighting` section of your `solrconfig.xml` file, adjusting the type, language, and country values as appropriate to your application:

```
<boundaryScanner name="breakIterator"
class="solr.highlight.BreakIteratorBoundaryScanner">
  <lst name="defaults">
    <str name="hl.bs.type">WORD</str>
    <str name="hl.bs.language">en</str>
    <str name="hl.bs.country">US</str>
  </lst>
</boundaryScanner>
```

Possible values for the `hl.bs.type` parameter are WORD, LINE, SENTENCE, and CHARACTER.

#### *The `simple` Boundary Scanner*

The `simple` boundary scanner scans term boundaries for a specified maximum character value (`hl.bs.maxScan`) and for common delimiters such as punctuation marks (`hl.bs.chars`). The `simple` boundary scanner may be useful for some custom To implement the `simple` boundary scanner, add this code to the `highlighting` section of your `solrconfig.xml` file, adjusting the values as appropriate to your application:

```
<boundaryScanner name="simple" class="solr.highlight.SimpleBoundaryScanner"
default="true">
  <lst name="defaults">
    <str name="hl.bs.maxScan">10</str>
    <str name="hl.bs.chars">.,!?\t\n</str>
  </lst>
</boundaryScanner>
```

#### Related Content

- HighlightingParameters from the Solr wiki
- Highlighting javadocs

## Postings Highlighter

PostingsHighlighter focuses on good document summarizes and efficiency, but is less flexible than the other highlighters. It uses significantly less disk space, focuses on good document summaries, and provides a performant approach if queries have a low number of terms relative to the number of results per page. However, the drawbacks are that it is not a query matching debugger (it focuses on fast highlighting for full-text search) and it does not allow broken analysis chains.

To use this highlighter, you must turn on `storeOffsetsWithPositions` for the field. There is no need to turn on `termVectors`, `termPositions`, or `termOffsets` in fields since this highlighter does not make use of term vectors.

#### Configuring Postings Highlighter

The configuration for the Postings Highlighter is done in `solrconfig.xml`.

First, define the search component:

```
<searchComponent class="solr.HighlightComponent" name="highlight">
  <highlighting class="org.apache.solr.highlight.PostingsSolrHighlighter"/>
</searchComponent>
```

Note in this example, we have named the search component "highlight". If you started with a default `solrconfig.xml` file, then you already have a component with that name. You should either replace the default with this example, or rename the search component that is already there so there is no confusion about which search component implementation Solr should use.

Then in the request handler, you can define the defaults, as in this example:

```
<requestHandler name="standard" class="solr.StandardRequestHandler">
  <lst name="defaults">
    <int name="hl.snippets">1</int>
    <str name="hl.tag.pre">&lt;em&gt;</str>
    <str name="hl.tag.post">&lt;/em&gt;</str>
    <str name="hl.tag.ellipsis">... </str>
    <bool name="hl.defaultSummary">true</bool>
    <str name="hl.encoder">simple</str>
    <float name="hl.score.k1">1.2</float>
    <float name="hl.score.b">0.75</float>
    <float name="hl.score.pivot">87</float>
    <str name="hl.bs.language"></str>
    <str name="hl.bs.country"></str>
    <str name="hl.bs.variant"></str>
    <str name="hl.bs.type">SENTENCE</str>
    <int name="hl.maxAnalyzedChars">10000</int>
  </lst>
</requestHandler>
```

This example shows all of the defaults for each parameter. If you intend to keep all of the defaults, you would not need to add anything to the request handler and could override the default values at query time as needed.

### Postings Highlighter Parameters

The table below describes Solr's parameters for this highlighter. These parameters can be set as defaults (as in the examples), or the default values can be changed in the request handler or at query time. Most of the parameters can be specified per-field (exceptions noted below).

| Parameter | Default | Description |
|---|---|---|
| hl | blank (no highlight) | 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. |
| hl.q | blank | Specifies an overriding query term for highlighting. If `hl.q` is specified, the highlighter will use that term rather than the main query term. |

| | | |
|---|---|---|
| hl.fl | blank | 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.<br>A '*' can be used to match field globs, such as 'text_*' or even '*' to highlight on all fields where highlighting is possible. When using '*', consider adding `hl.requireFieldMatch=true`. |
| hl.snippets | 1 | Specifies maximum number of highlighted snippets to generate per field. It is possible for any number of snippets from zero to this value to be generated. This parameter accepts per-field overrides. |
| hl.tag.pre | <em> | Specifies the text that should appear before a highlighted term. |
| hl.tag.post | </em> | Specifies the text that should appear after a highlighted term. |
| hl.tag.ellipsis | "... " | Specifies the text that should join two unconnected passages in the resulting snippet. |
| hl.maxAnalyzedChars | 10000 | Specifies the number of characters into a document that Solr should look for suitable snippets. This parameter does not accept per-field overrides. |
| hl.multiValuedSeparatorChar | " " (space) | Specifies the logical separator between multi-valued fields. |
| hl.defaultSummary | true | If **true**, a field should have a default summary if highlighting finds no matching passages. |
| hl.encoder | simple | Defines the encoding for the resulting snippet. The value **simple** applies no escaping, while **html** will escape HTML characters in the text. |
| hl.score.k1 | 1.2 | Specifies BM25 term frequency normalization parameter 'k1'. For example, it can be set to "0" to rank passages solely based on the number of query terms that match. |
| hl.score.b | 0.75 | Specifies BM25 length normalization parameter 'b'. For example, it can be set to "0" to ignore the length of passages entirely when ranking. |
| hl.score.pivot | 87 | Specifies BM25 average passage length in characters. |
| hl.bs.language | blank | Specifies the breakiterator language for dividing the document into passages. |
| hl.bs.country | blank | Specifies the breakiterator country for dividing the document into passages. |
| hl.bs.variant | blank | Specifies the breakiterator variant for dividing the document into passages. |

| | | |
|---|---|---|
| hl.bs.type | SENTENCE | Specifies the breakiterator type for dividing the document into passages. Can be **SENTENCE**, **WORD**, **CHARACTER**, **LINE**, or **WHOLE**. |

**Related Content**

- PostingsHighlighter from the Solr wiki
- PostingsSolrHighlighter javadoc

# Spell Checking

The SpellCheck component is designed to provide inline query suggestions based on other, similar, terms. The basis for these suggestions can be terms in a field in Solr, externally created text files, or fields in other Lucene indexes.

Topics covered in this section:

- Configuring the SpellCheckComponent
- Spell Check Parameters
- Distributed SpellCheck

## Configuring the SpellCheckComponent

### Define Spell Check in `solrconfig.xml`

The first step is to specify the source of terms in `solrconfig.xml`. There are three approaches to spell checking in Solr, discussed below.

#### *IndexBasedSpellChecker*

The `IndexBasedSpellChecker` uses a Solr index as the basis for a parallel index used for spell checking. It requires defining a field as the basis for the index terms; a common practice is to copy terms from some fields (such as `title`, `body`, etc.) to another field created for spell checking. Here is a simple example of configuring `solrconfig.xml` with the `IndexBasedSpellChecker`:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="classname">solr.IndexBasedSpellChecker</str>
    <str name="spellcheckIndexDir">./spellchecker</str>
    <str name="field">content</str>
    <str name="buildOnCommit">true</str>
  </lst>
</searchComponent>
```

The first element defines the `searchComponent` to use the `solr.SpellCheckComponent`. The `classname` is the specific implementation of the SpellCheckComponent, in this case `solr.IndexBasedSpellChecker`. Defining the `classname` is optional; if not defined, it will default to `IndexBasedSpellChecker`.

The `spellcheckIndexDir` defines the location of the directory that holds the spellcheck index, while the `field` defines the source field (defined in `schema.xml`) for spell check terms. When choosing a field for the spellcheck index, it's best to avoid a heavily processed field to get more accurate results. If the field has many word variations from processing synonyms and/or stemming, the dictionary will be created with those variations in addition to more valid spelling data.

Finally, *buildOnCommit* defines whether to build the spell check index at every commit (that is, every time new documents are added to the index). It is optional, and can be omitted if you would rather set it to `false`.

#### *DirectSolrSpellChecker*

The `DirectSolrSpellChecker` uses terms from the Solr index without building a parallel index like the `IndexBasedSpellChecker`. It is considered experimental and still in development, but is being used widely. This spell checker has the benefit of not having to be built regularly, meaning that the terms are always up-to-date with terms in the index. Here is how this might be configured in `solrconfig.xml`

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">name</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    <str name="distanceMeasure">internal</str>
    <float name="accuracy">0.5</float>
    <int name="maxEdits">2</int>
    <int name="minPrefix">1</int>
    <int name="maxInspections">5</int>
    <int name="minQueryLength">4</int>
    <float name="maxQueryFrequency">0.01</float>
    <float name="thresholdTokenFrequency">.01</float>
  </lst>
</searchComponent>
```

When choosing a `field` to query for this spell checker, you want one which has relatively little analysis performed on it (particularly analysis such as stemming). Note that you need to specify a field to use for the suggestions, so like the `IndexBasedSpellChecker`, you may want to copy data from fields like `title`, `body`, etc., to a field dedicated to providing spelling suggestions.

Many of the parameters relate to how this spell checker should query the index for term suggestions. The `distanceMeasure` defines the metric to use during the spell check query. The value "internal" uses the default Levenshtein metric, which is the same metric used with the other spell checker implementations.

Because this spell checker is querying the main index, you may want to limit how often it queries the index to be sure to avoid any performance conflicts with user queries. The `accuracy` setting defines the threshold for a valid suggestion, while `maxEdits` defines the number of changes to the term to allow. Since most spelling mistakes are only 1 letter off, setting this to 1 will reduce the number of possible suggestions (the default, however, is 2); the value can only be 1 or 2. `minPrefix` defines the minimum number of characters the terms should share. Setting this to 1 means that the spelling suggestions will all start with the same letter, for example.

The `maxInspections` parameter defines the maximum number of possible matches to review before returning results; the default is 5. `minQueryLength` defines how many characters must be in the query before suggestions are provided; the default is 4. `maxQueryFrequency` sets the maximum threshold for the number of documents a term must appear in before being considered as a suggestion. This can be a percentage (such as .01, or 1%) or an absolute value (such as 4). A lower threshold is better for small indexes. Finally, `tresholdTokenFrequency` sets the minimum number of documents a term must appear in, and can also be expressed as a percentage or an absolute value.

#### *FileBasedSpellChecker*

The `FileBasedSpellChecker` uses an external file as a spelling dictionary. This can be useful if using Solr as a spelling server, or if spelling suggestions don't need to be based on actual terms in the index. In `solrconfig.xml`, you would define the searchComponent as so:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="classname">solr.FileBasedSpellChecker</str>
    <str name="name">file</str>
    <str name="sourceLocation">spellings.txt</str>
    <str name="characterEncoding">UTF-8</str>
    <str name="spellcheckIndexDir">./spellcheckerFile</str>
  </lst>
</searchComponent>
```

The differences here are the use of the `sourceLocation` to define the location of the file of terms and the use of `characterEncoding` to define the encoding of the terms file.

> ⓘ In the previous example, *name* is used to name this specific definition of the spellchecker. Multiple definitions can co-exist in a single `solrconfig.xml`, and the *name* helps to differentiate them when they are defined in the `schema.xml`. If only defining one spellchecker, no name is required.

#### *WordBreakSolrSpellChecker*

`WordBreakSolrSpellChecker` offers suggestions by combining adjacent query terms and/or breaking terms into multiple words. It is a `SpellCheckComponent` enhancement, leveraging Lucene's `WordBreakSpellChecker`. It can detect spelling errors resulting from misplaced whitespace without the use of shingle-based dictionaries and provides collation support for word-break errors, including cases where the user has a mix of single-word spelling errors and word-break errors in the same query. It also provides shard support.

Here is how it might be configured in `solrconfig.xml`:

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="name">wordbreak</str>
    <str name="classname">solr.WordBreakSolrSpellChecker</str>
    <str name="field">lowerfilt</str>
    <str name="combineWords">true</str>
    <str name="breakWords">true</str>
    <int name="maxChanges">10</int>
  </lst>
</searchComponent>
```

Some of the parameters will be familiar from the discussion of the other spell checkers, such as `name`, `classname`, and `field`. New for this spell checker is `combineWords`, which defines whether words should be combined in a dictionary search (default is true); `breakWords`, which defines if words should be broken during a dictionary search (default is true); and `maxChanges`, an integer which defines how many times the spell checker should check collation possibilities against the index (default is 10).

The spellchecker can be configured with a traditional checker (ie: `DirectSolrSpellChecker`). The results are combined and collations can contain a mix of corrections from both spellcheckers.

#### Add It to a Request Handler

Queries will be sent to a RequestHandler. If every request should generate a suggestion, then you would add the following to the `requestHandler` that you are using:

```
<str name="spellcheck">true</str>
```

One of the possible parameters is the `spellcheck.dictionary` to use, and multiples can be defined. With multiple dictionaries, all specified dictionaries are consulted and results are interleaved. Collations are created with combinations from the different spellcheckers, with care taken that multiple overlapping corrections do not occur in the same collation.

Here is an example with multiple dictionaries:

```
<requestHandler name="spellCheckWithWordbreak"
class="org.apache.solr.handler.component.SearchHandler">
  <lst name="defaults">
    <str name="spellcheck.dictionary">default</str>
    <str name="spellcheck.dictionary">wordbreak</str>
    <str name="spellcheck.count">20</str>
  </lst>
  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

Until Solr v4.7, the only way to provide auto-complete (or search term suggestions based on user query input) was to extend the SpellCheckComponent. There is now a preferred approach using the SuggestComponent, described in the section Suggester.

If you are still using the SpellCheckComponent-based approach, we recommend changing to the SuggestComponent instead. However, the below snippet from solrconfig.xml that shows how to configure `solrconfig.xml` to use the SpellCheckComponent.

```
<searchComponent class="solr.SpellCheckComponent" name="suggest">
   <lst name="spellchecker">
      <str name="name">suggest</str>
      <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
      <str
name="lookupImpl">org.apache.solr.spelling.suggest.tst.TSTLookup</str>
      <str name="field">name</str>  <!-- the indexed field to derive suggestions
from -->
      <float name="threshold">0.005</float>
      <str name="buildOnCommit">true</str>
<!--
      <str name="sourceLocation">american-english</str>
-->
   </lst>
</searchComponent>
```

See the Suggester section for the available options for the `lookupImpl` property. Note also that using the

SpellCheckComponent for auto-suggest does not support the DirectSolrSpellChecker; an index of terms must always be created in order to serve suggestions.

### Spell Check Parameters

The SpellCheck component accepts the parameters described in the table below. All of these parameters can be overridden by specifying `spellcheck.collateParam.xx` where *xx* is the parameter you are overriding.

| Parameter | Description |
|---|---|
| spellcheck | Turns on or off SpellCheck suggestions for the request. If **true**, then spelling suggestions will be generated. |
| spellcheck.q or 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.maxCollations | This parameter specifies the maximum number of collations to return. |
| spellcheck.maxCollationTries | This parameter specifies the number of collation possibilities for Solr to try before giving up. |
| spellcheck.maxCollationEvaluations | This parameter specifies the maximum number of word correction combinations to rank and evaluate prior to deciding which collation candidates to test against the index. |
| spellcheck.collateExtendedResult | If true, returns an expanded response detailing the collations found. If `spellcheck.collate` is false, this parameter will be ignored. |
| spellcheck.collateMaxCollectDocs | The maximum number of documents to collect when testing potential Collations |
| 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.maxResultsForSuggest | The maximum number of hits the request can return in order to both generate spelling suggestions and set the "correctlySpelled" element to "false". |

| | |
|---|---|
| spellcheck.alternativeTermCount | The count of suggestions to return for each query term existing in the index and/or dictionary. |
| spellcheck.reload | Reloads the spellchecker. |
| spellcheck.accuracy | Specifies an accuracy value to help decide whether a result is worthwhile. |
| spellcheck.<DICT_NAME>.key | Specifies a key/value pair for the implementation handling a given dictionary. |

#### The `spellcheck` Parameter

This parameter turns on SpellCheck suggestions for the request. If **true**, then spelling suggestions will be generated.

#### The `spellcheck.q` or `q` Parameter

This 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, and so on. 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.

> ⚠ 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.

#### 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.

The dictionary will take some time to build, so this parameter should not be sent with every request.

#### 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.

#### 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.

#### The `spellcheck.onlyMorePopular` Parameter

If **true**, Solr will to return suggestions that result in more hits for the query than the existing query. Note that this will return more popular suggestions even when the given query term is present in the index and considered "correct".

#### The `spellcheck.maxResultsForSuggest` Parameter

For example, if this is set to 5 and the user's query returns 5 or fewer results, the spellchecker will report "correctlySpelled=false" and also offer suggestions (and collations if requested). Setting this greater than zero is useful for creating "did-you-mean?" suggestions for queries that return a low number of hits.

#### The `spellcheck.alternativeTermCount` Parameter

Specify the number of suggestions to return for each query term existing in the index and/or dictionary. Presumably, users will want fewer suggestions for words with docFrequency>0. Also setting this value turns "on" context-sensitive spell suggestions.

#### The `spellcheck.extendedResults` Parameter

This parameter causes to Solr to include additional information about the suggestion, such as the frequency in the index.

#### The `spellcheck.collate` Parameter

If **true**, this parameter directs Solr to take the best suggestion for each token (if one 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".

The spellcheck.collate parameter only returns collations that are guaranteed to result in hits if re-queried, even when applying original `fq` parameters. This is especially helpful when there is more than one correction per query.

> ⚠ This only returns a query to be used. It does not actually run the suggested query.

#### The `spellcheck.maxCollations` Parameter

The maximum number of collations to return. The default is **1**. This parameter is ignored if `spellcheck.collate` i s false.

#### The `spellcheck.maxCollationTries` Parameter

This parameter specifies the number of collation possibilities for Solr to try before giving up. Lower values ensure better performance. Higher values may be necessary to find a collation that can return results. The default value is `0` , which maintains backwards-compatible (Solr 1.4) behavior (do not check collations). This parameter is ignored if `s pellcheck.collate` is false.

#### The `spellcheck.maxCollationEvaluations` Parameter

This parameter specifies the maximum number of word correction combinations to rank and evaluate prior to deciding which collation candidates to test against the index. This is a performance safety-net in case a user enters a query with many misspelled words. The default is **10,000** combinations, which should work well in most situations.

#### The `spellcheck.collateExtendedResult` Parameter

If **true**, this parameter returns an expanded response format detailing the collations Solr found. The default value is **f alse** and this is ignored if `spellcheck.collate` is false.

#### The `spellcheck.collateMaxCollectDocs` Parameter

This parameter specifies the maximum number of documents that should be collect when testing potential collations against the index. A value of **0** indicates that all documents should be collected, resulting in exact hit-counts.

Otherwise an estimation is provided as a performance optimization in cases where exact hit-counts are unnecessary – the higher the value specified, the more precise the estimation.

The default value for this parameter is **0**, but when `spellcheck.collateExtendedResults` is **false**, the optimization is always used as if a **1** had been specified.

### 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.

### The `spellcheck.accuracy` Parameter

Specifies an accuracy value to be used by the spell checking implementation to decide whether a result is worthwhile or not. The value is a float between 0 and 1. Defaults to `Float.MIN_VALUE`.

### The `spellcheck.<DICT_NAME>.key` Parameter

Specifies a key/value pair for the implementation handling a given dictionary. The value that is passed through is just `key=value` (`spellcheck.<DICT_NAME>.` is stripped off.

For example, given a dictionary called `foo`, `spellcheck.foo.myKey=myValue` would result in `myKey=myValue` being passed through to the implementation handling the dictionary `foo`.

### 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>
```

### Distributed SpellCheck

The `SpellCheckComponent` also supports spellchecking on distributed indexes. If you are using the SpellCheckComponent on a request handler other than "/select", you must provide the following two parameters:

| Parameter | Description |
|---|---|
| shards | Specifies the shards in your distributed indexing configuration. For more information about distributed indexing, see Distributed Search with Index Sharding |
| shards.qt | Specifies the request handler Solr uses for requests to shards. This parameter is not required for the `/select` request handler. |

For example: `http://localhost:8983/solr/select?q=*:*&spellcheck=true&spellcheck.build=true&spellcheck.q=toyata&qt=spell&shards.qt=spell&shards=solr-shard1:8983/solr,solr-shard2:8983/solr`

In case of a distributed request to the SpellCheckComponent, the shards are requested for at least five suggestions even if the `spellcheck.count` parameter value is less than five. Once the suggestions are collected, they are ranked by the configured distance measure (Levenstein Distance by default) and then by aggregate frequency.

## Query Re-Ranking

Query Re-Ranking allows you to run a simple query (A) for matching documents and then re-rank the top N documents using the scores from a more complex query (B). Since the more costly ranking from query B is only applied to the top N documents it will have less impact on performance then just using the complex query B by itself – the trade off is that documents which score very low using the simple query A may not be considered during the re-ranking phase, even if they would score very highly using query B.

### Specifying A Ranking Query

A Ranking query can be specified using the "`rq`" request parameter.  The "rq" parameter must specify a query string that when parsed, produces a RankQuery. This could also be done with a custom QParserPlugin you have written as a plugin, but most users can just use the "`rerank`" parser provided with Solr.

The "`rerank`" parser wraps a query specified by an local parameter, along with additional parameters indicating how many documents should be re-ranked, and how the final scores should be computed:

| Parameter | Default | Description |
|---|---|---|
| `reRankQuery` | (Mandatory) | The query string for your complex ranking query - in most cases a variable will be used to refer to another request parameter. |
| `reRankDocs` | 200 | The number of top N documents from the original query that should be re-ranked. This number will be treated as a minimum, and may be increased internally automatically in order to rank enough documents to satisfy the query (ie: start+rows) |
| `reRankWeight` | 2.0 | A multiplicative factor that will be applied to the score from the reRankQuery for each of the top matching documents, before that score is added to the original score |

In the example below, the top 1000 documents matching the query "greetings" will be re-ranked using the query "(hi hello hey hiya)". The resulting scores for each of those 1000 documents will be 3 times their score from the "(hi hello hey hiya)", plus the score from the original "gretings" query:

```
q=greetings&rq={!rerank reRankQuery=$rqq reRankDocs=1000
reRankWeight=3}&rqq=(hi+hello+hey+hiya)
```

If a document matches the original query, but does not match the re-ranking query, the document's original score will remain.

### Result Paging With Query Re-Ranking

Query Re-Ranking is designed to be used with result pages that fall within the re-ranked document window (reRankDocs). After paging beyond reRankDocs, the Re-Ranking parameters should be dropped from the query. For example if reRankDocs is 1000 and the "start" parameter is 1001, then Re-Ranking should not be applied.

For example, in the query below the page displayed falls within the reRankDocs, so Re-Ranking is applied.

```
q=greetings&start=201&rows=20&rq={!rerank reRankQuery=$rqq reRankDocs=1000
reRankWeight=3}&rqq=(hi+hello+hey+hiya)
```

After paging beyond reRankDocs the Re-Ranking should be removed from the query:

```
q=greetings&start=1001&rows=20
```

### Combining Ranking Queries With Other Solr Features

The "`rq`" parameter and the re-ranking feature in general works well with other Solr features. For example, it can be used in conjunction with the collapse parser to re-rank the group heads after they've been collapsed. It also preserves the order of documents elevated by the elevation component. And it even has it's own custom explain so you can see how the re-ranking scores were derived when looking at debug information.

## Transforming Result Documents

Document Transformers can be used to modify the information returned about each documents in the results of a query.

### Using Document Transformers

When executing a request, a document transformer can be used by including it in the `fl` parameter using square brackets, for example:

```
fl=id,name,score,[shard]
```

Some transformers allow, or require, local parameters which can be specified as key value pairs inside the brackets:

```
fl=id,name,score,[explain style=nl]
```

As with regular fields, you can change the key used when a Transformer adds a field to a document via a prefix:

```
fl=id,name,score,my_val_a:[value v=42 t=int],my_val_b:[value v=7 t=float]
```

The sections below discuss exactly what these various transformers do.

### Available Transformers

#### `[value]` - ValueAugmenterFactory

Modifies every document to include the exact same value, as if it were a stored field in every document:

```
q=*:*&fl=id,greeting:[value v='hello']
```

The above query would produce results like the following:

```
<result name="response" numFound="32" start="0">
  <doc>
    <str name="id">1</str>
    <str name="greeting">hello</str></doc>
  </doc>
  ...
```

By default, values are returned as a String, but a "`t`" parameter can be specified using a value of int, float, double, or date to force a specific return type:

```
q=*:*&fl=id,my_number:[value v=42 t=int],my_string:[value v=42]
```

In addition to using these request parameters, you can configure additional named instances of ValueAugmenterFactory, or override the default behavior of the existing `[value]` transformer in your solrconfig.xml file:

```
<transformer name="mytrans2"
class="org.apache.solr.response.transform.ValueAugmenterFactory" >
  <int name="value">5</int>
</transformer>
<transformer name="value"
class="org.apache.solr.response.transform.ValueAugmenterFactory" >
  <double name="defaultValue">5</double>
</transformer>
```

The "`value`" option forces an explicit value to always be used, while the "`defaultValue`" option provides a default that can still be overridden using the "`v`" and "`t`" local parameters.

#### `[explain]` - ExplainAugmenterFactory

Augments each document with an inline explanation of it's score exactly like the information available about each document in the debug section:

```
q=features:cache&wt=json&fl=id,[explain style=nl]
```

Supported values for "`style`" are "`text`", and "`html`", and "nl" which returns the information as structured data:

```
"response":{"numFound":2,"start":0,"docs":[
      {
        "id":"6H500F0",
        "[explain]":{
          "match":true,
          "value":1.052226,
          "description":"weight(features:cache in 2) [DefaultSimilarity], result of:",
          "details":[{
...
```

A default style can be configured by specifying an "args" parameter in your configuration:

```
<transformer name="explain"
class="org.apache.solr.response.transform.ExplainAugmenterFactory" >
  <str name="args">nl</str>
</transformer>
```

### `[child]` - ChildDocTransformerFactory

This transformer returns all descendant documents of each parent document matching your query in a flat list nested inside the matching parent document. This is useful when you have indexed nested child documents and want to retrieve the child documents for the relevant parent documents for any type of search query.

```
fl=id,[child parentFilter=doc_type:book childFilter=doc_type:chapter limit=100]
```

Note that this transformer can be used even though the query itself is not a Block Join query.

When using this transformer, the `parentFilter` parameter must be specified, and works the same as in all Block Join Queries, additional optional parameters are:

- `childFilter` - query to filter which child documents should be included, this can be particularly useful when you have multiple levels of hierarchical documents (default: all children)
- `limit` - the maximum number of child documents to be returned per parent document (default: 10)

### `[shard]` - ShardAugmenterFactory

This transformer adds information about what shard each individual document came from in a distributed request.

ShardAugmenterFactory does not support any request parameters, or configuration options.

### `[docid]` - DocIdAugmenterFactory

This transformer adds the internal Lucene document id to each document – this is primarily only useful for debugging purposes.

DocIdAugmenterFactory does not support any request parameters, or configuration options.

### `[elevated]` and `[excluded]`

These transformers are available only when using the Query Elevation Component.

- `[elevated]` annotates each document to indicate if it was elevated or not.
- `[excluded]` annotates each document to indicate if it would have been excluded - this is only supported if you also use the `markExcludes` parameter.

```
fl=id,[elevated],[excluded]&excludeIds=GB18030TEST&elevateIds=6H500F0&markExcludes=tru
e
```

```
"response":{"numFound":32,"start":0,"docs":[
      {
        "id":"6H500F0",
        "[elevated]":true,
        "[excluded]":false},
      {
        "id":"GB18030TEST",
        "[elevated]":false,
        "[excluded]":true},
      {
        "id":"SP2514N",
        "[elevated]":false,
        "[excluded]":false},
...
```

## Suggester

The SuggestComponent in Solr provides users with automatic suggestions for query terms. You can use this to implement a powerful auto-suggest feature in your search application.

Solr has long had the autosuggest functionality, but Solr 4.7 introduced a new approach based on a dedicated SuggestComponent. This approach utilizes Lucene's Suggester implementation and supports all of the lookup implementations available in Lucene.

The main features of this Suggester are:

- Lookup implementation pluggability
- Term dictionary pluggability, giving you the flexibility to choose the dictionary implementation
- Distributed support

The `solrconfig.xml` found in Solr's `example` directory has the new Suggester implementation configured already. If you would like to switch to this approach from a legacy system, you will need to modify your current implementation to add a search component and a request handler, described below (for more on search components, see the section Request Handlers and SearchComponents in SolrConfig).

Covered in this section:

### Configuring Suggester in solrconfig.xml

The example `solrconfig.xml` found in `./example/solr/collection1/conf` has a Suggester searchComponent and a /suggest requestHandler already configured. You can use that as the basis for your configuration, or create it from scratch, as detailed below.

#### Adding the Suggest Search Component

In the example `solrconfig.xml` found in The first step is to add a search component to `solrconfig.xml` to extend the SpellChecker. Here is some sample code that could be used.

```
<searchComponent name="suggest" class="solr.SuggestComponent">
   <lst name="suggester">
      <str name="name">mySuggester</str>
      <str name="lookupImpl">FuzzyLookupFactory</str>
      <str name="dictionaryImpl">DocumentDictionaryFactory</str>
      <str name="field">cat</str>
      <str name="weightField">price</str>
      <str name="suggestAnalyzerFieldType">string</str>
   </lst>
</searchComponent>
```

##### *Suggester Search Component Parameters*

The Suggester search component takes several configuration parameters. The choice of the lookup implementation (`lookupImpl`, how terms are found in the suggestion dictionary) and the dictionary implementation (`dictionaryImpl`, how terms are stored in the suggestion dictionary) will dictate some of the parameters required. Below are the main parameters that can be used no matter what lookup or dictionary implementation is used. In the following sections additional parameters are provided for each implementation.

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>searchComponent name</td><td>Arbitrary name for the search component.</td></tr>
<tr><td>name</td><td>A symbolic name for this suggester. You can refer to this name in the URL parameters and in the SearchHandler configuration. It is possible to have mutiples of these</td></tr>
<tr><td>lookupImpl</td><td>Lookup implementation. There are several possible implementations, described below in the section Lookup Implementations.</td></tr>
<tr><td>dictionaryImpl</td><td>The dictionary implementation to use. There are several possible implementations, described below in the section Dictionary Implementations.</td></tr>
<tr><td>field</td><td>A field from the index to use as the basis of suggestion terms. If <code>sourceLocation</code> is empty (meaning any dictionary implementation other than FileDictionaryFactory) then terms from this field in the index will be used.<br><br>To be used as the basis for a suggestion, the field must be indexed. You may want to use copyField rules to create a special 'suggest' field comprised of terms from other fields in documents. In any event, you likely want a minimal amount of analysis on the field, so an additional option is to create a field type in your schema that only uses basic tokenizers or filters. One option for such a field type is shown here:<br><pre>
&lt;fieldType class="solr.TextField" name="textSpell"
positionIncrementGap="100"&gt;
   &lt;analyzer&gt;
      &lt;tokenizer class="solr.StandardTokenizerFactory"/&gt;
      &lt;filter class="solr.StandardFilterFactory"/&gt;
      &lt;filter class="solr.LowerCaseFilterFactory"/&gt;
   &lt;/analyzer&gt;
&lt;/fieldType&gt;
</pre>However, this minimal analysis is not required if you want more analysis to occur on terms. If using the AnalyzingLookupFactory as your lookupImpl, however, you have the option of defining the field type rules to use for index and query time analysis.</td></tr>
<tr><td>sourceLocation</td><td>The path to the dictionary file if using the FileDictionaryFactory. If this value is empty then the main index will be used as a source of terms and weights.</td></tr>
<tr><td>storeDir</td><td>The location to store the dictionary file.</td></tr>
<tr><td>buildOnCommit or buildOnOptimize</td><td>If true then the lookup data structure will be rebuilt after commit. If false, the default, then the lookup data will be built only when requested by URL parameter <code>suggest.build=true</code>. Use <code>buildOnCommit</code> to rebuild the dictionary with every commit, or <code>buildOnOptimize</code> to build the dictionary only when the index is optimized.</td></tr>
</table>

### ***Lookup Implementations***

The `lookupImpl` parameter defines the algorithms used to look up terms in the suggest index. There are several possible implementations to choose from, and some require additional parameters to be configured.

#### AnalyzingLookupFactory

A lookup that first analyzes the incoming text and adds the analyzed form to a weighted FST, and then does the same thing at lookup time.

This implementation uses the following additional properties:

- suggestAnalyzerFieldType: The field type to use for the query-time and build-time term suggestion analysis.
- exactMatchFirst: If true, the default, exact suggestions are returned first, even if they are prefixes or other strings in the FST have larger weights.
- preserveSep: If true, the default, then a separator between tokens is preserved. This means that suggestions are sensitive to tokenization (e.g., baseball is different from base ball).
- preservePositionIncrements: If true, the suggester will preserve position increments. This means that token filters which leave gaps (for example, when StopFilter matches a stopword) the position would be respected when building the suggester. The default is false.

#### FuzzyLookupFactory

This is a suggester which is an extension of the AnalyzingSuggester but is fuzzy in nature. The similarity is measured by the Levenshtein algorithm.

This implementation uses the following additional properties:

- exactMatchFirst: If true, the default, exact suggestions are returned first, even if they are prefixes or other strings in the FST have larger weights.
- preserveSep: If true, the default, then a separator between tokens is preserved. This means that suggestions are sensitive to tokenization (e.g., baseball is different from base ball).
- maxSurfaceFormsPerAnalyzedForm: Maximum number of surface forms to keep for a single analyzed form. When there are too many surface forms we discard the lowest weighted ones.
- maxGraphExpansions: When building the FST ("index-time"), we add each path through the tokenstream graph as an individual entry. This places an upper-bound on how many expansions will be added for a single suggestion. The default is -1 which means there is no limit.
- preservePositionIncrements: If true, the suggester will preserve position increments. This means that token filters which leave gaps (for example, when StopFilter matches a stopword) the position would be respected when building the suggester. The default is false.
- maxEdits: The maximum number of string edits allowed. The systems hard limit is 2. The default is 1.
- transpositions: If true, the default, transpositions should be treated as a primitive edit operation.
- nonFuzzyPrefix: The length of the common non fuzzy prefix match which must match a suggestion. The default is 1.
- minFuzzyLength: The minimum length of query before which any string edits will be allowed. The default is 3.
- unicodeAware: If true, maxEdits, minFuzzyLength, transpositions and nonFuzzyPrefix parameters will be measured in unicode code points (actual letters) instead of bytes. The default is false.

#### AnalyzingInfixSuggesterFactory

Analyzes the input text and then suggests matches based on prefix matches to any tokens in the indexed text. This uses a lucene index for it's dictionary.

This implementation uses the following additional properties.

- indexPath: When using AnalyzingInfixSuggester you can provide your own path where the index will get built.

The default is analyzingInfixSuggesterIndexDir and will be created in your collections data directory.

- minPrefixChars: Minimum number of leading characters before PrefixQuery is used (default is 4). Prefixes shorter than this are indexed as character ngrams (increasing index size but making lookups faster).

#### BlendedInfixLookupFactory

An extension of the AnalyzingInfixSuggester which provides additional functionality to weight prefix matches across the matched documents. You can tell it to score higher if a hit is closer to the start of the suggestion or vice versa.

This implementation uses the following additional properties:

- blenderType: used to calculate weight coefficient using the position of the first matching word.
- linear: weightFieldValue*(1 - 0.10*position): Matches to the start will be given a higher score (Default)
- reciprocal: weightFieldValue/(1+position): Matches to the end will be given a higher score.
- numFactor: The factor to multiply the number of searched elements from which results will be pruned. Default is 10.
- indexPath: When using BlendedInfixSuggester you can provide your own path where the index will get built. The default directory name is blendedInfixSuggesterIndexDir and will be created in your collections data directory.
- minPrefixChars: Minimum number of leading characters before PrefixQuery is used (default 4). Prefixes shorter than this are indexed as character ngrams (increasing index size but making lookups faster).

#### FreeTextSuggesterFactory

It looks at the last tokens plus the prefix of whatever final token the user is typing, if present, to predict the most likely next token. The number of previous tokens that need to be considered can also be specified. This suggester would only be used as a fallback, when the primary suggester fails to find any suggestions.

This implementation uses only one additional parameter:

- ngrams: The max number of tokens out of which singles will be make the dictionary. The default value is 2. Increasing this would mean you want more than the previous 2 tokens to be taken into consideration when making the suggestions.

#### FSTLookupFactory

An automaton-based lookup. This implementation is slower to build, but provides the lowest memory cost. We recommend using this implementation unless you need more sophisticated matching results, in which case you should use the Jaspell implementation.

This implementation uses the following additional properties:

- exactMatchFirst: If true, the default, exact suggestions are returned first, even if they are prefixes or other strings in the FST have larger weights.
- weightBuckets: The number of separate buckets for weights which the suggester will use while building it's dictionary.

#### TSTLookupFactory

A simple compact ternary trie based lookup.

#### WFSTLookupFactory

A weighted automaton representation which is an alternative to FSTLookup for more fine-grained ranking. WFSTLookup does not use buckets, but instead a shortest path algorithm. Note that it expects weights to be whole numbers. If weight is missing it's assumed to be 1.0. Weights affect the sorting of matching suggestions when `spel` `lcheck.onlyMorePopular=true` is selected: weights are treated as "popularity" score, with higher weights preferred over suggestions with lower weights.

#### JaspellLookupFactory

A more complex lookup based on a ternary trie from the JaSpell project. Use this implementation if you need more sophisticated matching results.

### *Dictionary Implementations*

The dictionary implementations define how terms are stored. There are several options, and multiple dictionaries can be used in a single request if necessary.

#### DocumentDictionaryFactory

A dictionary with terms, weights, and an optional payload taken from the index.

This dictionary implementation takes the following parameters in addition to parameters described for the Suggester generally and for the lookup implementation:

- weightField: A field that is stored or a numeric DocValue field. This field is required.
- payloadfield: The payloadField should be a field that is stored. This field is optional.

#### DocumentExpressionFactory

This dictionary implementation is the same as the DocumentDictionaryFactory but allows users to specify an arbitrary expression into the 'weightExpression' tag.

This dictionary implementation takes the following parameters in addition to parameters described for the Suggester generally and for the lookup implementation:

- weightField: A field that is stored or a numeric DocValue field. This field is required.
- payloadfield: The payloadField should be a field that is stored. This field is optional.
- weightExpression: An arbitrary expression used for scoring the suggestions. The fields used must be numeric fields.

#### HighFrequencyDictionaryFactory

This dictionary implementation allows adding a threshold to prune out less frequent terms in cases where very common terms may overwhelm other terms.

This dictionary implementation takes one parameter in addition to parameters described for the Suggester generally and for the lookup implementation:

- threshold: A value between zero and one representing the minimum fraction of the total documents where a term should appear in order to be added to the lookup dictionary.

#### FileDictionaryFactory

This dictionary implementation allows using an external file that contains suggest entries. Weights and payloads can also be used.

If using a dictionary file, it should be a plain text file in UTF-8 encoding. Blank lines and lines that start with a '#' are ignored. You can use both single terms and phrases in the dictionary file. If adding weights or payloads, those should be separated from terms using the delimiter defined with the `fieldDelimiter` property (the default is '\t', the tab representation).

This dictionary implementation takes one parameter in addition to parameters described for the Suggester generally and for the lookup implementation:

- fieldDelimiter: Specify the delimiter to be used separating the entries, weights and payloads. The default is tab ('\t').

```
# This is a sample dictionary file.

acquire
accidentally\t2.0
accommodate\t3.0
```

#### *Multiple Dictionaries*

It is possible to include multiple dictionaryImpl definitions in a single SuggestComponent definition.

To do this, simply define separate suggesters, as in this example:

```
<searchComponent name="suggest" class="solr.SuggestComponent">
   <lst name="suggester">
      <str name="name">suggester1</str>
      <str name="lookupImpl">FuzzyLookupFactory</str>
      <str name="dictionaryImpl">DocumentDictionaryFactory</str>
      <str name="field">cat</str>
      <str name="weightField">price</str>
      <str name="suggestAnalyzerFieldType">string</str>
   </lst>
   <lst name="suggester">
      <str name="name">suggester2 </str>
      <str name="dictionaryImpl">DocumentExpressionDictionaryFactory</str>
      <str name="lookupImpl">FuzzyLookupFactory</str>
      <str name="field">product_name</str>
      <str name="weightExpression">((price * 2) + ln(popularity))</str>
      <str name="sortField">weight</str>
      <str name="sortField">price</str>
      <str name="storeDir">suggest_fuzzy_doc_expr_dict</str>
      <str name="suggestAnalyzerFieldType">text_en</str>
   </lst>
</searchComponent>
```

When using these Suggesters in a query, you would define multiple 'suggest.dictionary' parameters in the request, referring to the names given for each Suggester in the search component definition. The response will include the terms in sections for each Suggester. See the Examples section below for an example request and response.

#### Adding the Suggest Request Handler

After adding the search component, a request handler must be added to `solrconfig.xml`. This request handler works the same as any other request handler, and allows you to configure default parameters for serving suggestion requests. The request handler definition must incorporate the "suggest" search component defined previously.

```
<requestHandler name="/suggest" class="solr.SearchHandler" startup="lazy">
    <lst name="defaults">
      <str name="suggest">true</str>
      <str name="suggest.count">10</str>
    </lst>
    <arr name="components">
      <str>suggest</str>
    </arr>
  </requestHandler>
```

#### *Suggest Request Handler Parameters*

The following parameters allow you to set defaults for the Suggest request handler:

| Parameter | Description |
|---|---|
| suggest=true | This parameter should always be true, because we always want to run the Suggester for queries submitted to this handler. |
| suggest.dictionary | The name of the dictionary component configured in the search component. This is a mandatory parameter. It can be set in the request handler, or sent as a parameter at query time. |
| suggest.q | The query to use for suggestion lookups. |
| suggest.count | Specifies the number of suggestions for Solr to return. |
| suggest.build | If true, it will build the suggester index. This is likely useful only for initial requests; you would probably not want to build the dictionary on every request, particularly in a production system. If you would like to keep your dictionary up to date, you should use the `buildOnCommit` or `buildOnOptimize` parameter for the search component. |
| suggest.reload | If true, it will reload the suggester index. |
| buildAll | If true, it will build all suggester indexes. |
| reloadAll | If true, it will reload all suggester indexes. |

These properties can also be overridden at query time, or not set in the request handler at all and always sent at query time.

## Example Usages

### Get Suggestions with Weights

This is the basic suggestion using a single dictionary and a single Solr core.

Example query:

```
http://localhost:8983/solr/suggest?suggest=true&suggest.build=true&suggest.dictionary=
suggest1&suggest.q=elec&wt=json
```

In this example, we've simply requested the string 'elec' with the suggest.q parameter and requested that the suggestion dictionary be built with suggest.build (note, however, that you would likely not want to build the index on every query - instead you should use buildOnCommit or buildOnOptimize if you have regularly changing documents).

Example response:

```
{
    "responseHeader": {
        "status": 0,
        "QTime": 35
    },
    "command": "build",
    "suggest": {
        "suggest1": {
            "elec": {
                "numFound": 3,
                "suggestions": [
                    {
                        "term": "electronics and computer1",
                        "weight": 2199,
                        "payload": ""
                    },
                    {
                        "term": "electronics",
                        "weight": 649,
                        "payload": ""
                    },
                    {
                        "term": "electronics and stuff2",
                        "weight": 279,
                        "payload": ""
                    }
                ]
            }
        }
    }
}
```

### Suggestions in a Distributed System

It is possible to get suggestions in SolrCloud mode, using the shards.qt parameter:

```
http://localhost:8983/solr/suggest?suggest.dictionary=suggester2&suggest=true&suggest.
build=true&suggest.q=elec&shards=localhost:8983/solr,localhost:7574/solr&shards.qt=/su
ggest
```

### Multiple Dictionaries

If you have defined multiple dictionaries, you can use them in queries.

Example query:

```
http://localhost:8983/solr/suggest?suggest=true&suggest.dictionary=suggester1&suggest.
dictionary=suggester2&suggest.q=elec
```

In this example we have sent the string 'elec' as the suggest.q parameter and named two suggest.dictionary definitions to be used.

Example response:

```
{
    responseHeader: {
        status: 0,
        QTime: 3
    },
    suggest: {
        suggester1: {
            e: {
                numFound: 1,
                suggestions: [
                    {
                        term: "electronics and computer1",
                        weight: 100,
                        payload: ""
                    }
                ]
            }
        },
        suggester2: {
            e: {
                numFound: 1,
                suggestions: [
                    {
                        term: "electronics and computer1",
                        weight: 10,
                        payload: ""
                    }
                ]
            }
        }
    }
}
```

## MoreLikeThis

The `MoreLikeThis` search component enables users to query for documents similar to a document in their result list. It does this by using terms from the original document to find similar documents in the index.

Covered in this section:

There are three ways to use MoreLikeThis. The first, and most common, is to use it as a request handler. In this case, you would send text to the MoreLikeThis request handler as needed (as in when a user clicked on a "similar documents" link). The second is to use it as a search component. This is less desirable since it performs the MoreLikeThis analysis on every document returned. This may slow search results. The final approach is to use it as a request handler but with externally supplied text. This case, also referred to as the MoreLikeThisHandler, will supply information about similar documents in the index based on the text of the input document.

### How MoreLikeThis Works

`MoreLikeThis` constructs a Lucene query based on terms in a document. It does this by pulling terms from the defined list of fields ( see the `mlt.fl` parameter, below). For best results, the fields should have stored term vectors in `schema.xml`. For example:

```
<field name="cat" ... termVectors="true" />
```

If term vectors are not stored, `MoreLikeThis` will generate terms from stored fields. A `uniqueKey` must also be stored in order for MoreLikeThis to work properly.

The next phase filters terms from the original document using thresholds defined with the MoreLikeThis parameters. Finally, a query is run with these terms, and any other query parameters that have been defined (see the `mlt.qf` parameter, below) and a new document set is returned.

> In Solr 4.1, MoreLikeThis supports distributed search.

### Common Parameters for MoreLikeThis

The table below summarizes the `MoreLikeThis` parameters supported by Lucene/Solr. These parameters can be used with any of the three possible MoreLikeThis approaches.

| Parameter | Description |
|---|---|
| mlt.fl | Specifies the fields to use for similarity. If possible, these should have stored `termVectors`. |
| mlt.mintf | Specifies the Minimum Term Frequency, the frequency below which terms will be ignored in the source document. |
| mlt.mindf | Specifies the Minimum Document Frequency, the frequency at which words will be ignored which do not occur in at least this many documents. |
| mlt.maxdf | Specifies the Maximum Document Frequency, the frequency at which words will be ignored which occur in more than this many documents. **New in Solr 4.1** |
| 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 | Specifies if the query will be boosted by the interesting term relevance. It can be either "true" or "false". |
| mlt.qf | Query fields and their boosts using the same format as that used by the DisMaxRequestHandler. These fields must also be specified in `mlt.fl`. |

### Parameters for the MoreLikeThisComponent.

Using MoreLikeThis as a search component returns similar documents for each document in the response set. In addition to the common parameters, these additional options are available:

| 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. |

### Parameters for the MoreLikeThisHandler

The table below summarizes parameters accessible through the `MoreLikeThisHandler`. It supports faceting, paging, and filtering using common query parameters, but does not work well with alternate query parsers.

| 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`. |

### Related Topics

- RequestHandlers and SearchComponents in SolrConfig

# Pagination of Results

### Basic Pagination

In most search application usage, the "top" matching results (sorted by score, or some other criteria) are then displayed to some human user.  In many applications the UI for these sorted results are displayed to the user in "pages" containing a fixed number of matching results, and users don't typically look at results past the first few pages worth of results.

In Solr, this basic paginated searching is supported using the **`start`** and **`rows`** parameters, and performance of this common behaviour can be tuned by utilizing the `queryResultCache` and adjusting the `queryResultWindowSize` configuration options based on your expected page sizes.

#### Basic Pagination Examples

The easiest way to think about simple pagination, is to simply multiply the page number you want (treating the "first" page number as "0") by the number of rows per page; such as in the following psuedo-code:

```
function fetch_solr_page($page_number, $rows_per_page) {
  $start = $page_number * $rows_per_page
  $params = [ q = $some_query, rows = $rows_per_page, start = $start ]
  return fetch_solr($params)
}
```

#### How Basic Pagination is Affected by Index Updates

The `start` param specified in a request to Solr indicates an **absolute** "offset" in the complete sorted list of matches that the client wants Solr to use as the beginning of the current "page".  If an index modification (such as adding or removing documents) which affects the sequence of ordered documents matching a query occurs in between two requests from a client for subsequent pages of results, then it is possible that these modifications can result in the same document being returned on multiple pages, or documents being "skipped" as the result set shrinks or grows.

For example: consider an index containing 26 documents like so:

| id | name |
|---|---|
| 1 | A |
| 2 | B |
| ... | |
| 26 | Z |

Followed by the following requests & index modifications interleaved:

- A client requests `q=*:*&rows=5&start=0&sort=name asc`
  - documents with the ids `1-5` will be returned to the client in
- Document id `3` is deleted
- The client requests "page #2" using `q=*:*&rows=5&start=5&sort=name asc`
  - Documents `7-11` will be returned
  - Document `6` has been skipped, since it is now the 5th document in the sorted set of all matching results, and would be returned on a new request for "page #1"
- 3 new documents are now added with the ids `90`, `91`, and `92`; All three documents have a name of `A`
- The client requests "page #3" using `q=*:*&rows=5&start=10&sort=name asc`
  - Documents `9-13` will be returned
  - Documents `9`, `10`, and `11` have now been returned on both page #2 and page #3 since they moved farther back in the list of sorted results

In typical situations these impacts from index changes on paginated searching don't significantly affect user experience -- either because they happen extremely infrequently in fairly static collections, or because the users recognize that the collection of data is constantly evolving and expect to see documents shift up in down in the result sets.

#### Performance Problems with "Deep Paging"

In some situations, the results of a Solr search are not destined for a simple paginated user interface.  When you wish to fetch a very large number of sorted results from Solr to feed into an external system, using very large values for the `start` or `rows` parameters can be very inefficient. Pagination using `start` and `rows` not only require Solr to compute (and sort) in memory all of the matching documents that should be fetched for the current page, but also all of the documents that would have appeared on previous pages. So while a request for `start=0&rows=1000000` may be obviously inefficient because it requires Solr to maintain & sort in memory a set of 1 million documents, likewise a request for `start=999000&rows=1000` is equally inefficient for the same reasons. Solr can't compute which matching document is the 999001st result in sorted order, without first determining what the first 999000 matching sorted results are.

## Fetching A Large Number of Sorted Results: Cursors

As an alternative to increasing the "start" parameter to request subsequent pages of sorted results, Solr supports using a "Cursor" to scan through results. Cursors in Solr are a logical concept, that doesn't involve caching any state information on the server. Instead the sort values of the last document returned to the client are used to compute a "mark" representing a logical point in the ordered space of sort values. That "mark" can be specified in the parameters of subsequent requests to tell Solr where to continue.

### Using Cursors

To use a cursor with Solr, specify a **`cursorMark`** parameter with the value of `"*"`. You can think of this being analogous to `start=0` as a way to tell Solr "start at the beginning of my sorted results" except that it also informs Solr that you want to use a Cursor. So in addition to returning the top N sorted results (where you can control N using the `rows` parameter) the Solr response will also include an encoded String named **`nextCursorMark`**. You then take the `nextCursorMark` String value from the response, and pass it back to Solr as the `cursorMark` parameter for your next request. You can repeat this process until you've fetched as many docs as you want, or until the `nextCursorMark` returned matches the `cursorMark` you've already specified -- indicating that there are no more results.

### Constraints when using Cursors

There are a few important constraints to be aware of when using `cursorMark` parameter in a Solr request

1. `cursorMark` and `start` are mutually exclusive parameters
    - Your requests must either not include a `start` parameter, or it must be specified with a value of "`0`".
2. `sort` clauses must include the uniqueKey field (either "`asc`" or "`desc`")
    - If `id` is your uniqueKey field, then sort params like `id asc` and `name asc, id desc` would both work fine, but `name asc` by itself would not

Cursor mark values are computed based on the sort values of each document in the result, which means multiple documents with identical sort values will produce identical Cursor mark values if one of them is the last document on a page of results. In that situation, the subsequent request using that `cursorMark` would not know which of the documents with the identical mark values should be skipped. Requiring that the uniqueKey field be used as a clause in the sort criteria guarantees that a deterministic ordering will be returned, and that every `cursorMark` value will identify a unique point in the sequence of documents.

### Cursor Examples

#### *Fetch All Docs*

The psuedo-code shown here shows the basic logic involved in fetching all documents matching a query using a cursor:

```
// when fetching all docs, you might as well use a simple id sort
// unless you really need the docs to come back in a specific order
$params = [ q => $some_query, sort => 'id asc', rows => $r, cursorMark => '*' ]
$done = false
while (not $done) {
  $results = fetch_solr($params)
  // do something with $results
  if ($params[cursorMark] == $results[nextCursorMark]) {
    $done = true
  }
  $params[cursorMark] = $results[nextCursorMark]
}
```

Using SolrJ, this psuedo-code would be:

```
SolrQuery q = (new SolrQuery(some_query)).setRows(r).setSort(SortClause.asc("id"));
String cursorMark = CursorMarkParams.CURSOR_MARK_START;
boolean done = false;
while (! done) {
  q.set(CursorMarkParams.CURSOR_MARK_PARAM, cursorMark);
  QueryResponse rsp = solrServer.query(q);
  String nextCursorMark = rsp.getNextCursorMark();
  doCustomProcessingOfResults(rsp);
  if (cursorMark.equals(nextCursorMark)) {
    done = true;
  }
  cursorMark = nextCursorMark;
}
```

If you wanted to do this by hand using curl, the sequence of requests would look something like this:

```
$ curl '...&rows=10&sort=id+asc&cursorMark=*'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 docs here ...
  ]},
  "nextCursorMark":"AoEjR0JQ"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEjR0JQ'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 more docs here ...
  ]},
  "nextCursorMark":"AoEpVkRCREIxQTE2"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEpVkRCREIxQTE2'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 10 more docs here ...
  ]},
  "nextCursorMark":"AoEmbWF4dG9y"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEmbWF4dG9y'
{
  "response":{"numFound":32,"start":0,"docs":[
    // ... 2 docs here because we've reached the end.
  ]},
  "nextCursorMark":"AoEpdmlld3Nvbmlj"}
$ curl '...&rows=10&sort=id+asc&cursorMark=AoEpdmlld3Nvbmlj'
{
  "response":{"numFound":32,"start":0,"docs":[
    // no more docs here, and note that the nextCursorMark
    // matches the cursorMark param we used
  ]},
  "nextCursorMark":"AoEpdmlld3Nvbmlj"}
```

#### ***Fetch first N docs, Based on Post Processing***

Since the cursor is stateless from Solr's perspective, your client code can stop fetching additional results as soon as you have decided you have enough information:

```
 while (! done) {
  q.set(CursorMarkParams.CURSOR_MARK_PARAM, cursorMark);
  QueryResponse rsp = solrServer.query(q);
  String nextCursorMark = rsp.getNextCursorMark();
  boolean hadEnough = doCustomProcessingOfResults(rsp);
  if (hadEnough || cursorMark.equals(nextCursorMark)) {
    done = true;
  }
  cursorMark = nextCursorMark;
}
```

#### **How cursors are Affected by Index Updates**

Unlike basic pagination, Cursor pagination does not rely on using an absolute "offset" into the completed sorted list of matching documents.  Instead, the `cursorMark` specified in a request encapsulates information about the **relative** position of the last document returned, based on the **absolute** sort values of that document.  This means that the impact of index modifications is much smaller when using a cursor compared to basic pagination.

Consider the same example index described when discussing basic pagination:

| id | name |
|---|---|
| 1 | A |
| 2 | B |
| ... | |
| 26 | Z |

- A client requests `q=*:*&rows=5&start=0&sort=name asc, id asc&cursorMark=*`
    - Documents with the ids `1-5` will be returned to the client in order
- Document id `3` is deleted
- The client requests 5 more documents using the `nextCursorMark` from the previous response
    - Documents `6-10` will be returned -- the deletion of a document that's already been returned doesn't affect the relative position of the cursor
- 3 new documents are now added with the ids `90`, `91`, and `92`; All three documents have a name of `A`
- The client requests 5 more documents using the `nextCursorMark` from the previous response
    - Documents `11-15` will be returned -- the addition of new documents with sort values already past does not affect the relative position of the cursor
- Document id `1` is updated to change it's 'name' to `Q`
- Document id 17 is updated to change it's 'name' to `A`
- The client requests 5 more documents using the `nextCursorMark` from the previous response
    - The resulting documents are `16,1,18,19,20` in that order
    - Because the sort value of document `1` changed so that it is *after* the cursor position, the document is returned to the client twice
    - Because the sort value of document `17` changed so that it is *before* the cursor position, the document has been "skipped" and will not be returned to the client as the cursor continues to progress

In a nutshell: When fetching all results matching a query using `cursorMark`, the only way index modifications can result in a document being skipped, or returned twice, is if the sort value of the document changes.

> One way to ensure that a document will never be returned more then once, is to use the uniqueKey field as the primary (and therefore: only significant) sort criteria.
>
> In this situation, you will be guaranteed that each document is only returned once, no matter how it may be be modified during the use of the cursor.

### "Tailing" a Cursor

Because Cursor requests are stateless, and the `cursorMark` values encapsulate the **absolute** sort values of the last document returned from a search, it's possible to "continue" fetching additional results from a cursor that has already reached it's end -- if new documents are added (or existing documents are updated) to the end of the results.  You can think of this as similar to using something like `"tail -f"` in Unix.

The most common examples of how this can be useful is when you have a "timestamp" field recording when a document has been added/updated in your index.  Client applications can continuously poll a cursor using a `sort=timestamp asc, id asc` for documents matching a query, and always be notified when a document is added or updated matching the request criteria.  Another common example is when you have uniqueKey values that always increase as new documents are created, and you can continuously poll a cursor using `sort=id asc` to be notified about new documents.

The psuedo-code for tailing a cursor is only a slight modification from our early example for processing all docs matching a query:

```
while (true) {
  $doneForNow = false
  while (not $doneForNow) {
    $results = fetch_solr($params)
    // do something with $results
    if ($params[cursorMark] == $results[nextCursorMark]) {
      $doneForNow = true
    }
    $params[cursorMark] = $results[nextCursorMark]
  }
  sleep($some_configured_delay)
}
```

## Result Grouping

Result Grouping groups documents with a common field value into groups and returns the top documents for each group. For example, if you searched for "DVD" on an electronic retailer's e-commerce site, you might be returned three categories such as "TV and Video," "Movies," and "Computers," with three results per category. In this case, the query term "DVD" appeared in all three categories, so Solr groups them together in order to increase relevancy for the user.

Result Grouping is separate from Faceting. Though it is conceptually similar, faceting returns all relevant results and allows the user to refine the results based on the facet category. For example, if you searched for "shoes" on a footwear retailer's e-commerce site, you would be returned all results for that query term, along with selectable facets such as "size," "color," "brand," and so on.

However, with Solr 4 you can also group facets. The grouped faceting works with the first `group.field` parameter, and other `group.field` parameters are ignored.

Grouped faceting supports `facet.field` and `facet.range` but currently doesn't support date and pivot faceting.

Grouped faceting differs from non grouped facets (sum of all facets) == (total of products with that property) as shown in the following example:

Object 1

- name: Phaser 4620a
- ppm: 62
- product_range: 6

Object 2

- name: Phaser 4620i
- ppm: 65

- product_range: 6

Object 3

- name: ML6512
- ppm: 62
- product_range: 7

If you ask Solr to group these documents by "product_range", then the total amount of groups is 2, but the facets for ppm are 2 for 62 and 1 for 65.

### Request Parameters

Result Grouping takes the following request parameters. Any number of these request parameters can be included in a single request:

| Parameter | Type | Description |
|---|---|---|
| group | Boolean | If true, query results will be grouped. |
| group.field | string | The name of the field by which to group results. The field must be single-valued, and either be indexed or a field type that has a value source and works in a function query, such as `ExternalFileField`. It must also be a string-based field, such as `StrField` or `TextField` |
| group.func | query | Group based on the unique values of a function query. Supported since Solr 4.0. |
| group.query | query | Return a single group of documents that match the given query. |
| rows | integer | The number of groups to return. The default value is 10. |
| start | integer | Specifies an initial offset for the list of groups. |
| group.limit | integer | Specifies the number of results to return for each group. The default value is 1. |
| group.offset | integer | Specifies an initial offset for the document list of each group. |
| sort | sortspec | Specifies how Solr sorts the groups relative to each other. For example, `sort=popularity desc` will cause the groups to be sorted according to the highest popularity document in each group. The default value is `s core desc`. |
| group.sort | sortspec | Specifies how Solr sorts documents within a single group. The default value is `score desc`. |
| group.format | grouped/simple | If this parameter is set to `simple`, the grouped documents are presented in a single flat list, and the `start` and `rows` parameters affect the numbers of documents instead of groups. |
| group.main | Boolean | If true, the result of the first field grouping command is used as the main result list in the response, using `group.format=simple`. |

| | | |
|---|---|---|
| group.ngroups | Boolean | If true, Solr includes the number of groups that have matched the query in the results. The default value is false. |
| group.truncate | Boolean | If true, facet counts are based on the most relevant document of each group matching the query. The default value is false. |
| group.facet | Boolean | Determines whether to compute grouped facets for the field facets specified in facet.field parameters. Grouped facets are computed based on the first specified group. As with normal field faceting, fields shouldn't be tokenized (otherwise counts are computed for each token). Grouped faceting supports single and multivalued fields. Default is false. New with Solr 4. |
| group.cache.percent | integer between 0 and 100 | Setting this parameter to a number greater than 0 enables caching for result grouping. Result Grouping executes two searches; this option caches the second search. The default value is 0. Testing has shown that group caching only improves search time with Boolean, wildcard, and fuzzy queries. For simple queries like term or "match all" queries, group caching degrades performance. |

Any number of group commands (`group.field`, `group.func`, `group.query`) may be specified in a single request.

Grouping is also supported for distributed searches. Currently `group.func` is the only parameter that doesn't supported distributed searches.

## Examples

All of the following examples work with the data provided in the Solr Example directory.

### Grouping Results by Field

In this example, we will group results based on the `manu_exact` field, which specifies the manufacturer of the items in the sample dataset.

```
http://localhost:8983/solr/select?wt=json&indent=true&fl=id,name&q=solr+memory&group=true&group.field=manu_exact
```

```
{
...
"grouped":{
  "manu_exact":{
    "matches":6,
    "groups":[{
        "groupValue":"Apache Software Foundation",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
              "id":"SOLR1000",
              "name":"Solr, the Enterprise Search Server"}]
        }},
      {
        "groupValue":"Corsair Microsystems Inc.",
        "doclist":{"numFound":2,"start":0,"docs":[
            {
              "id":"VS1GB400C3",
              "name":"CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC
3200) System Memory - Retail"}]
        }},
      {
        "groupValue":"A-DATA Technology Inc.",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
              "id":"VDBDB1A16",
              "name":"A-DATA V-Series 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC
3200) System Memory - OEM"}]
        }},
      {
        "groupValue":"Canon Inc.",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
              "id":"0579B002",
              "name":"Canon PIXMA MP500 All-In-One Photo Printer"}]
        }},
      {
        "groupValue":"ASUS Computer Inc.",
        "doclist":{"numFound":1,"start":0,"docs":[
            {
              "id":"EN7800GTX/2DHTV/256M",
              "name":"ASUS Extreme N7800GTX/2DHTV (256 MB)"}]
        }
      }
    ]
  }
}
```

The response indicates that there are six total matches for our query. For each unique value of `group.field`, Solr returns a `docList` with the top scoring document. The `docList` also includes the total number of matches in that group as the `numFound` value. The groups are sorted by the score of the top document within each group.

We can run the same query with the request parameter `group.main=true`. This will format the results as a single flat document list. This flat format does not include as much information as the normal result grouping query results, but it may be easier for existing Solr clients to parse.

`http://localhost:8983/solr/select?wt=json&indent=true&fl=id,name,manufacturer&q=solr+`

memory&group=true&group.field=manu_exact&group.main=true

```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "fl":"id,name,manufacturer",
      "indent":"true",
      "q":"solr memory",
      "group.field":"manu_exact",
      "group.main":"true",
      "group":"true",
      "wt":"json"}},
  "grouped":{},
  "response":{"numFound":6,"start":0,"docs":[
      {
        "id":"SOLR1000",
        "name":"Solr, the Enterprise Search Server"},
      {
        "id":"VS1GB400C3",
        "name":"CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200)
System Memory - Retail"},
      {
        "id":"VDBDB1A16",
        "name":"A-DATA V-Series 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC 3200)
System Memory - OEM"},
      {
        "id":"0579B002",
        "name":"Canon PIXMA MP500 All-In-One Photo Printer"},
      {
        "id":"EN7800GTX/2DHTV/256M",
        "name":"ASUS Extreme N7800GTX/2DHTV (256 MB)"}]
  }
}
```

#### Grouping by Query

In this example, we will use the `group.query` parameter to find the top three results for "memory" in two different price ranges: 0.00 to 99.99, and over 100.

http://localhost:8983/solr/select?wt=json&indent=true&fl=name,price&q=memory&group=true&group.query=price:[0+TO+99.99]&group.query=price:[100+TO+*]&group.limit=3

```
{
  "responseHeader":{
    "status":0,
    "QTime":42,
    "params":{
      "fl":"name,price",
      "indent":"true",
      "q":"memory",
      "group.limit":"3",
      "group.query":["price:[0 TO 99.99]",
        "price:[100 TO *]"],
      "group":"true",
      "wt":"json"}},
  "grouped":{
    "price:[0 TO 99.99]":{
      "matches":5,
      "doclist":{"numFound":1,"start":0,"docs":[
          {
            "name":"CORSAIR ValueSelect 1GB 184-Pin DDR SDRAM Unbuffered DDR 400 (PC
3200) System Memory - Retail",
            "price":74.99}]
      }},
    "price:[100 TO *]":{
      "matches":5,
      "doclist":{"numFound":3,"start":0,"docs":[
          {
            "name":"CORSAIR  XMS 2GB (2 x 1GB) 184-Pin DDR SDRAM Unbuffered DDR 400
(PC 3200) Dual Channel
               Kit System Memory - Retail",
            "price":185.0},
          {
            "name":"Canon PIXMA MP500 All-In-One Photo Printer",
            "price":179.99},
          {
            "name":"ASUS Extreme N7800GTX/2DHTV (256 MB)",
            "price":479.95}]
      }
     }
   }
 }
```

In this case, Solr found five matches for "memory," but only returns four results grouped by price. This is because one result for "memory" did not have a price assigned to it.

### Distributed Result Grouping

Solr also supports result grouping on distributed indexes. If you are using result grouping on the "/select" request handler, you must provide the `shards` parameter described here. If you are using result grouping on a request handler other than "/select", you must also provide the `shards.qt` parameter:

| Parameter | Description |
|---|---|
| shards | Specifies the shards in your distributed indexing configuration. For more information about distributed indexing, see Distributed Search with Index Sharding |

| | |
|---|---|
| shards.qt | Specifies the request handler Solr uses for requests to shards. This parameter is not required for the `/select` request handler. |

For example: `http://localhost:8983/solr/select?wt=json&indent=true&fl=id,name,manufacturer&q=solr+memory&group=true&group.field=manu_exact&group.main=true&shards=solr-shard1:8983/solr,solr-shard2:8983/solr`

## Collapse and Expand Results

The collapsing query parser and the expand component combine to form an approach to grouping documents for field collapsing in search results.

### Collapsing Query Parser

The `CollapsingQParser` is really a *post filter* that provides more performant field collapsing than Solr's standard approach when the number of distinct groups in the result set is high. This parser collapses the result set to a single document per group before it forwards the result set to the rest of the search components. So all downstream components (faceting, highlighting, etc...) will work with the collapsed result set.

Collapse based on the highest scoring document:

```
fq={!collapse field=<field_name>}
```

Collapse based on the minimum value of a numeric field:

```
fq={!collapse field=<field_name> min=<field_name>}
```

Collapse based on the maximum value of a numeric field:

```
fq={!collapse field=<field_name> max=<field_name>}
```

Collapse based on the min/max value of a function. The **cscore()** function can be used with the CollapsingQParserPlugin to return the score of the current document being collapsed.

```
fq={!collapse field=<field_name> max=sum(cscore(),field(A))}
```

Collapse with a null policy:

```
fq={!collapse field=<field_name> nullPolicy=<nullPolicy>}
```

There are three null policies:

- **ignore**: removes documents with a null value in the collapse field. This is the default.
- **expand**: treats each document with a null value in the collapse field as a separate group.
- **collapse**: collapses all documents with a null value into a single group using either highest score, or minimum/maximum.

The CollapsingQParserPlugin fully supports the QueryElevationComponent.

### Expand Component

The ExpandComponent can be used to expand the groups that were collapsed by the CollapsingQParserPlugin.

Example usage with the CollapsingQParserPlugin:

```
q=foo&fq={!collapse field=ISBN}
```

In the query above, the CollapsingQParserPlugin will collapse the search results on the *ISBN* field. The main search results will contain the highest ranking document from each book.

The ExpandComponent can now be used to expand the results so you can see the documents grouped by ISBN. For example:

```
q=foo&fq={!collapse field=ISBN}&expand=true
```

The “expand=true” parameter turns on the ExpandComponent. The ExpandComponent adds a new section to the search output labeled “expanded”.

Inside the expanded section there is a *map* with each group head pointing to the expanded documents that are within the group. As applications iterate the main collapsed result set, they can access the *expanded* map to retrieve the expanded groups.

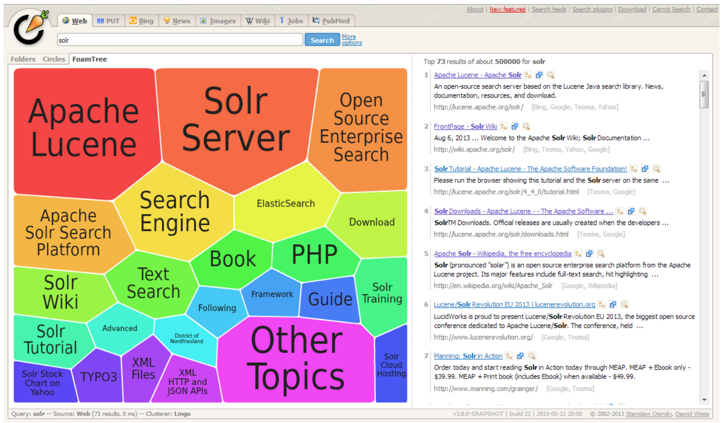
The ExpandComponent has the following parameters:

| **Parameter** | | **Default** |
|---|---|---|
| expand.sort | Orders the documents with the expanded groups | score desc |
| expand.rows | The number of rows to display in each group | 5 |
| expand.q | Overrides the main q parameter, determines which documents to include in the main group. | main q |
| expand.fq | Overrides main fq's, determines which documents to include in the main group. | main fq's |

## Result Clustering

The **clustering** (or **cluster analysis**) plugin attempts to automatically discover groups of related search hits (documents) and assign human-readable labels to these groups. By default in Solr, the clustering algorithm is applied to the search result of each single query—this is called an *on-line* clustering. While Solr contains an extension for for full-index clustering (*off-line* clustering) this section will focus on discussing on-line clustering only.

Clusters discovered for a given query can be perceived as *dynamic facets*. This is beneficial when regular faceting is difficult (field values are not known in advance) or when the queries are exploratory in nature. Take a look at the Carrot2 project's demo page to see an example of search results clustering in action (the groups in the visualization have been discovered automatically in search results to the right, there is no external information involved).


Web
PUT
Bing
News
Images
Wiki
Jobs
PubMed
About | New features! | Search feeds | Search plugins | Download | Carrot Search | Contact
Search
More options
Folders
Circles
FoamTree
Apache Lucene
Solr Server
Open Source Enterprise Search
Apache Solr Search Platform
Search Engine
ElasticSearch
Download
Book
PHP
Solr Wiki
Text Search
Framework
Guide
Solr Training
Following
Solr Tutorial
Advanced
District of Nordfriesland
Other Topics
Solr Cloud Hosting
Solr Stock Chart on Yahoo
TYPO3
XML Files
XML HTTP and JSON APIs
Query: solr -- Source: Web (73 results, 0 ms) -- Clusterer: Lingo
v3.8.0-SNAPSHOT | build 22 | 2013-05-22 20:00 © 2002-2013 Stanislaw Osinski, Dawid Weiss

solr

Top 73 results of about 500000 for solr

1 Apache Lucene - Apache Solr
An open-source search server based on the Lucene Java search library. News, documentation, resources, and download.
http://lucene.apache.org/solr/ [Bing, Google, Teoma, Yahoo]

2 FrontPage - Solr Wiki
Aug 6, 2013 ... Welcome to the Apache Solr Wiki; Solr Documentation ...
http://wiki.apache.org/solr/ [Bing, Teoma, Yahoo, Google]

3 Solr Tutorial - Apache Lucene - The Apache Software Foundation!
Please run the browser showing this tutorial and the Solr server on the same ...
http://lucene.apache.org/solr/4_4_0/tutorial.html [Teoma, Google]

4 Solr Downloads - Apache Lucene - - The Apache Software ...
SolrTM Downloads. Official releases are usually created when the developers ...
http://lucene.apache.org/solr/downloads.html [Teoma, Google]

5 Apache Solr - Wikipedia, the free encyclopedia
Solr (pronounced "solar") is an open source enterprise search platform from the Apache Lucene project. Its major features include full-text search, hit highlighting ...
http://en.wikipedia.org/wiki/Apache_Solr [Google, Wikipedia]

6 Lucene/Solr Revolution EU 2013 | lucenerevolution.org
LucidWorks is proud to present Lucene/Solr Revolution EU 2013, the biggest open source conference dedicated to Apache Lucene/Solr. The conference, held ...
http://www.lucenerevolution.org/ [Google, Teoma]

7 Manning: Solr in Action
Order today and start reading Solr in Action today through MEAP. MEAP + Ebook only - $39.99. MEAP + Print book (includes Ebook) when available - $49.99.
http://www.manning.com/grainger/ [Google, Teoma]

The query issued to the system was *Solr*. It seems clear that faceting could not yield a similar set of groups, although the goals of both techniques are similar—to let the user explore the set of search results and either rephrase the query or narrow the focus to a subset of current documents. Clustering is also similar to Result Grouping in that it can help to look deeper into search results, beyond the top few hits.

Topics covered in this section:

## Preliminary Concepts

Each **document** passed to the clustering component is composed of several logical parts:

- a unique identifier,
- origin URL,
- the title,
- the main content,
- a language code of the title and content.

The identifier part is mandatory, everything else is optional but at least one of the text fields (title or content) will be required to make the clustering process reasonable. It is important to remember that logical document parts must be mapped to a particular schema and its fields. The content (text) for clustering can be sourced from either a stored text field or context-filtered using a highlighter, all these options are explained below in the configuration section.

A **clustering algorithm** is the actual logic (implementation) that discovers relationships among the documents in the search result and forms human-readable cluster labels. Depending on the choice of the algorithm the clusters may (and probably will) vary. Solr comes with several algorithms implemented in the open source Carrot2 project, commercial alternatives also exist.

## Quick Start Example

Assuming an unpacked, unmodified distribution of Solr, issue the following commands in the console window:

```
cd example
java -Dsolr.clustering.enabled=true -jar start.jar
```

This command uses the same configuration and index as the main Solr example, but it additionally enables the clustering component contrib and a dedicated search handler configured to use it.

In a different console window, add some documents using the post tool (unless you have done so already):

```
cd example/exampledocs
java -jar post.jar *.xml
```

You can now try out the clustering handler by opening the following URL in a browser: `http://localhost:8983/solr/clustering?q=*:*&rows=100`

The output XML should include search hits and an array of automatically discovered clusters at the end, resembling the output shown here:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">299</int>
  </lst>
  <result name="response" numFound="32" start="0" maxScore="1.0">
    <doc>
      <str name="id">GB18030TEST</str>
      <str name="name">Test with some GB18030 encoded characters</str>
      <arr name="features">
        <str>No accents here</str>
        <str></str>
        <str>This is a feature (translated)</str>
        <str></str>
        <str>This document is very shiny (translated)</str>
      </arr>
      <float name="price">0.0</float>
      <str name="price_c">0,USD</str>
      <bool name="inStock">true</bool>
      <long name="_version_">1448955395025403904</long>
      <float name="score">1.0</float>
    </doc>

    <!-- more search hits, omitted -->
  </result>

  <arr name="clusters">
    <lst>
      <arr name="labels">
        <str>DDR</str>
      </arr>
      <double name="score">3.9599865057283354</double>
      <arr name="docs">
        <str>TWINX2048-3200PRO</str>
        <str>VS1GB400C3</str>
        <str>VDBDB1A16</str>
      </arr>
    </lst>
    <lst>
      <arr name="labels">
        <str>iPod</str>
      </arr>
      <double name="score">11.959228467119022</double>
      <arr name="docs">
        <str>F8V7067-APL-KIT</str>
        <str>IW-02</str>
        <str>MA147LL/A</str>
      </arr>
    </lst>

    <!-- More clusters here, omitted. -->

    <lst>
      <arr name="labels">
        <str>Other Topics</str>
      </arr>
      <double name="score">0.0</double>
      <bool name="other-topics">true</bool>
      <arr name="docs">
        <str>adata</str>
        <str>apple</str>
        <str>asus</str>
        <str>ati</str>
        <!-- other unassigned document IDs here -->
      </arr>
    </lst>
```

```
  </arr>
</response>
```

There were a few clusters discovered for this query (`*:*`), separating search hits into various categories: DDR, iPod, Hard Drive, etc. Each cluster has a label and score that indicates the "goodness" of the cluster. The score is algorithm-specific and is meaningful only in relation to the scores of other clusters in the same set. In other words, if cluster *A* has a higher score than cluster *B*, cluster *A* should be of better quality (have a better label and/or more coherent document set). Each cluster has an array of identifiers of documents belonging to it. These identifiers correspond to the `uniqueKey` field declared in the schema.

Depending on the quality of input documents, some clusters may not make much sense. Some documents may be left out and not be clustered at all; these will be assigned to the synthetic *Other Topics* group, marked with the `other-topics` property set to `true` (see the XML dump above for an example). The score of the other topics group is zero.

### Installation

The clustering contrib extension requires `dist/solr-clustering-*.jar` and all JARs under `contrib/clustering/lib`.

### Configuration

#### Declaration of the Search Component and Request Handler

Clustering extension is a search component and must be declared in `solrconfig.xml`. Such a component can be then appended to a request handler as the last component in the chain (because it requires search results which must be previously fetched by the search component).

An example configuration could look as shown below.

1. Include the required contrib JARs. Note paths are relative to the Solr core so they may need adjustments to your configuration.

   ```
   <lib dir="../../../contrib/clustering/lib/" regex=".*\.jar" />
   <lib dir="../../../dist/"                   regex="solr-clustering-\d.*\.jar" />
   ```

2. Declaration of the search component. Each component can also declare multiple clustering pipelines ("engines"), which can be selected at runtime.

```
<searchComponent name="clustering" class="solr.clustering.ClusteringComponent">
  <!-- Lingo clustering algorithm -->
  <lst name="engine">
    <str name="name">lingo</str>
    <str
name="carrot.algorithm">org.carrot2.clustering.lingo.LingoClusteringAlgorithm</st
r>
  </lst>

  <!-- An example definition for the STC clustering algorithm. -->
  <lst name="engine">
    <str name="name">stc</str>
    <str
name="carrot.algorithm">org.carrot2.clustering.stc.STCClusteringAlgorithm</str>
  </lst>
</searchComponent>
```

3. A request handler to which we append the clustering component declared above.

```
<requestHandler name="/clustering"
                class="solr.SearchHandler">
  <lst name="defaults">
    <bool name="clustering">true</bool>
    <bool name="clustering.results">true</bool>

    <!-- Logical field to physical field mapping. -->
    <str name="carrot.url">id</str>
    <str name="carrot.title">doctitle</str>
    <str name="carrot.snippet">content</str>

    <!-- Configure any other request handler parameters. We will cluster the
         top 100 search results so bump up the 'rows' parameter. -->
    <str name="rows">100</str>
    <str name="fl">*,score</str>
  </lst>

  <!-- Append clustering at the end of the list of search components. -->
  <arr name="last-components">
    <str>clustering</str>
  </arr>
</requestHandler>
```

#### Configuration Parameters of the Clustering Component

The table below summarizes parameters of each clustering engine or the entire clustering component (depending where they are declared).

| Parameter | Description |
|---|---|
| `clustering` | When `true`, clustering component is enabled. |
| `clustering.engine` | Declares which clustering engine to use. If not present, the first declared engine will become the default one. |

| | |
|---|---|
| `clustering.results` | When `true`, the component will perform clustering of search results (this should be enabled). |
| `clustering.collection` | When `true`, the component will perform clustering of the whole document index (this section does not cover full-index clustering). |

At the engine declaration level, the following parameters are supported.

| Parameter | Description |
|---|---|
| `carrot.algorithm` | The algorithm class. |
| `carrot.resourcesDir` | Algorithm-specific resources and configuration files (stop words, other lexical resources, default settings). By default points to `conf/clustering/carrot2/` |
| `carrot.outputSubClusters` | If `true` and the algorithm supports hierarchical clustering, sub-clusters will also be emitted. |
| `carrot.numDescriptions` | Maximum number of per-cluster labels to return (if the algorithm assigns more than one label to a cluster). |

The `carrot.algorithm` parameter should contain a fully qualified class name of an algorithm supported by the Carrot2 framework. Currently, the following algorithms are available:

- `org.carrot2.clustering.lingo.LingoClusteringAlgorithm` (open source)
- `org.carrot2.clustering.stc.STCClusteringAlgorithm` (open source)
- `org.carrot2.clustering.kmeans.BisectingKMeansClusteringAlgorithm` (open source)
- `com.carrotsearch.lingo3g.Lingo3GClusteringAlgorithm` (commercial)

For a comparison of characteristics of these algorithms see the following links:

- http://doc.carrot2.org/#section.advanced-topics.fine-tuning.choosing-algorithm
- http://project.carrot2.org/algorithms.html
- http://carrotsearch.com/lingo3g-comparison.html

The question of which algorithm to choose depends on the amount of traffic (STC is faster than Lingo, but arguably produces less intuitive clusters, Lingo3G is the fastest algorithm but is not free or open source), expected result (Lingo3G provides hierarchical clusters, Lingo and STC provide flat clusters), and the input data (each algorithm will cluster the input slightly differently). There is no one answer which algorithm is "the best".

#### Contextual and Full Field Clustering

The clustering engine can apply clustering to the full content of (stored) fields or it can run an internal highlighter pass to extract context-snippets before clustering. Highlighting is recommended when the logical snippet field contains a lot of content (this would affect clustering performance). Highlighting can also increase the quality of clustering because the content passed to the algorithm will be more focused around the query (it will be query-specific context). The following parameters control the internal highlighter.

| Parameter | Description |
|---|---|

| | |
|---|---|
| `carrot.produceSummary` | When `true` the clustering component will run a highlighter pass on the content of logical fields pointed to by `carrot.title` and `carrot.snippet`. Otherwise full content of those fields will be clustered. |
| `carrot.fragSize` | The size, in characters, of the snippets (aka fragments) created by the highlighter. If not specified, the default highlighting fragsize (`hl.fragsize`) will be used. |
| `carrot.summarySnippets` | The number of summary snippets to generate for clustering. If not specified, the default highlighting snippet count (`hl.snippets`) will be used. |

#### Logical to Document Field Mapping

As already mentioned in Preliminary Concepts, the clustering component clusters "documents" consisting of logical parts that need to be mapped onto physical schema of data stored in Solr. The field mapping attributes provide a connection between fields and logical document parts. Note that the content of title and snippet fields must be **stored** so that it can be retrieved at search time.

| Parameter | Description |
|---|---|
| `carrot.title` | The field (alternatively comma- or space-separated list of fields) that should be mapped to the logical document's title. The clustering algorithms typically give more weight to the content of the title field compared to the content (snippet). For best results, the field should contain concise, noise-free content. If there is no clear title in your data, you can leave this parameter blank. |
| `carrot.snippet` | The field (alternatively comma- or space-separated list of fields) that should be mapped to the logical document's main content. If this mapping points to very large content fields the performance of clustering may drop significantly. An alternative then is to use query-context snippets for clustering instead of full field content. See the description of the `carrot.produceSummary` parameter for details. |
| `carrot.url` | The field that should be mapped to the logical document's content URL. Leave blank if not required. |

#### Clustering Multilingual Content

The field mapping specification can include a `carrot.lang` parameter, which defines the field that stores ISO 639-1 code of the language in which the title and content of the document are written. This information can be stored in the index based on apriori knowledge of the documents' source or a language detection filter applied at indexing time. All algorithms inside the Carrot2 framework will accept ISO codes of languages defined in LanguageCode enum.

The language hint makes it easier for clustering algorithms to separate documents from different languages on input and to pick the right language resources for clustering. If you do have multi-lingual query results (or query results in a language different than English), it is strongly advised to map the language field appropriately.

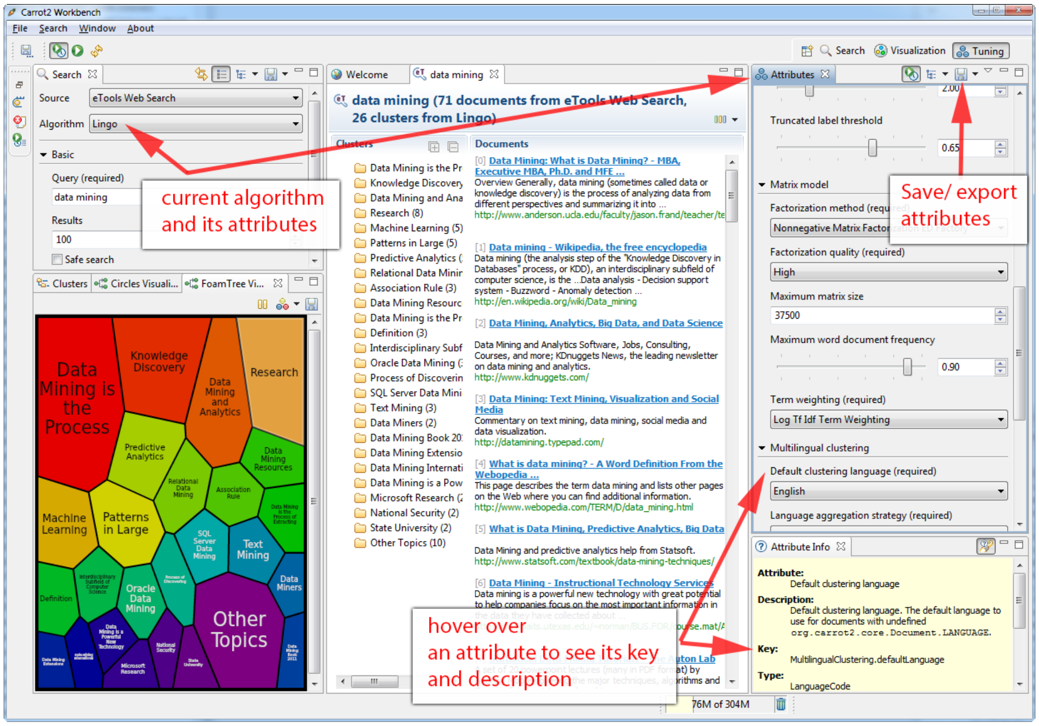
| Parameter | Description |
|---|---|
| `carrot.lang` | The field that stores ISO 639-1 code of the language of the document's text fields. |

| | |
|---|---|
| `carrot.lcmap` | A mapping of arbitrary strings into ISO 639 two-letter codes used by `carrot.lang`. The syntax of this parameter is the same as `langid.map.lcmap`, for example: `langid.map.lcmap=japanese:ja polish:pl english:en` |

The default language can also be set using Carrot2-specific algorithm attributes (in this case the MultilingualClustering.defaultLanguage attribute).

### Tweaking Algorithm Settings

The algorithms that come with Solr are using their default settings which may be inadequate for all data sets. All algorithms have lexical resources and resources (stop words, stemmers, parameters) that may require tweaking to get better clusters (and cluster labels). For Carrot2-based algorithms it is probably best to refer to a dedicated tuning application called Carrot2 Workbench (screenshot below). From this application one can export a set of algorithm attributes as an XML file, which can be then placed under the location pointed to by `carrot.resourcesDir`.



current algorithm and its attributes

Save/ export attributes

hover over an attribute to see its key and description

Source: eTools Web Search
Algorithm: Lingo
Query (required): data mining
Results: 100

data mining (71 documents from eTools Web Search, 26 clusters from Lingo)

Truncated label threshold: 0.65
Factorization method (required): Nonnegative Matrix Factorization
Factorization quality (required): High
Maximum matrix size: 37500
Maximum word document frequency: 0.90
Term weighting (required): Log Tf Idf Term Weighting
Default clustering language (required): English

Attribute: Default clustering language
Description: Default clustering language. The default language to use for documents with undefined org.carrot2.core.Document.LANGUAGE.
Key: MultilingualClustering.defaultLanguage
Type: LanguageCode

### Providing Defaults

The default attributes for all engines (algorithms) declared in the clustering component are placed under `carrot.resourcesDir` and with an expected file name of `engineName-attributes.xml`. So for an engine named `lingo` and the default value of `carrot.resourcesDir`, the attributes would be read from a file in `conf/clustering/carrot2/lingo-attributes.xml`.

An example XML file changing the default language of documents to Polish is shown below.

```
<attribute-sets default="attributes">
    <attribute-set id="attributes">
      <value-set>
        <label>attributes</label>
          <attribute key="MultilingualClustering.defaultLanguage">
            <value type="org.carrot2.core.LanguageCode" value="POLISH"/>
          </attribute>
      </value-set>
  </attribute-set>
</attribute-sets>
```

#### Tweaking at Query-Time

The clustering component and Carrot2 clustering algorithms can accept query-time attribute overrides. Note that certain things (for example lexical resources) can only be initialized once (at startup, via the XML configuration files).

An example query that changes the `LingoClusteringAlgorithm.desiredClusterCountBase` parameter for the Lingo algorithm: http://localhost:8983/solr/clustering?q=*:*&rows=100&LingoClusteringAlgorithm.desiredClusterCountBase=20

### Performance Considerations

Dynamic clustering of search results comes with two major performance penalties:

- Increased cost of fetching a larger-than-usual number of search results (50, 100 or more documents),
- Additional computational cost of the clustering itself.

For simple queries, the clustering time will usually dominate the fetch time. If the document content is very long the retrieval of stored content can become a bottleneck. The performance impact of clustering can be lowered in several ways:

- feed less content to the clustering algorithm by enabling `carrot.produceSummary` attribute,
- perform clustering on selected fields (titles only) to make the input smaller,
- use a faster algorithm (STC instead of Lingo, Lingo3G instead of STC),
- tune the performance attributes related directly to a specific algorithm.

Some of these techniques are described in *Apache SOLR and Carrot2 integration strategies* document, available at http://carrot2.github.io/solr-integration-strategies. The topic of improving performance is also included in the Carrot2 manual at http://doc.carrot2.org/#section.advanced-topics.fine-tuning.performance.

### Additional Resources

The following resources provide additional information about the clustering component in Solr and its potential applications.

- Apache Solr and Carrot2 integration strategies: http://carrot2.github.io/solr-integration-strategies
- Apache Solr Wiki (covers previous Solr versions, may be inaccurate): http://carrot2.github.io/solr-integration-strategies
- Clustering and Visualization of Solr search results (video from Berlin BuzzWords conference, 2011): http://vimeo.com/26616444

## Spatial Search

Solr supports location data for use in spatial/geospatial searches. Using spatial search, you can:

- Index points or other shapes
- Filter search results by a bounding box or circle or by other shapes
- Sort or boost scoring by distance between points, or relative area between rectangles
- Index and search multi-value time or other numeric durations

With Solr 4, there are three field types for spatial search: LatLonType (or its non-geodetic twin PointType), SpatialRecursivePrefixTreeFieldType (RPT for short), and BBoxField. LatLonType was the first spatial field, introduced in Solr 3; the others have been added since.  RPT offers more features than LatLonType and fast filter performance, although LatLonType is still more appropriate when efficient distance sorting/boosting is desired. They can both be used simultaneously for what each does best – LatLonType for sorting/boosting, RPT for filtering.  BBoxField is for indexing bounding boxes, querying by a box, specifying a search predicate (Intersects,IsWithin,Contains,DisjointTo), and a relevancy sort/boost like overlapRatio or simply the area.

For more information on Solr spatial search, see http://wiki.apache.org/solr/SpatialSearch.

### Indexing and Configuration

For indexing geodetic points (latitude and longitude), supply the pair of numbers as a string with a comma separating them in latitude then longitude order. For non-geodetic points, the order is x,y for PointType, and for RPT you must use a space instead of a comma, or use WKT.

See the section SpatialRecursivePrefixTreeFieldType below for RPT configuration specifics.

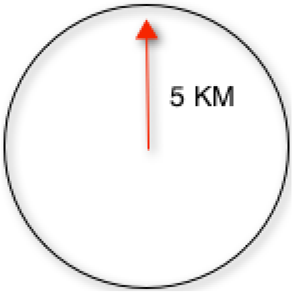
### Spatial Filters

The following parameters are used for spatial search:

| Parameter | Description |
|---|---|
| d | the radial distance, in kilometers (always; even for RPT field with units=degrees) |
| pt | the center point using the format "lat,lon" if latitude & longitude. Otherwise, "x,y" for PointType or "x y" for RPT field types. |
| sfield | a spatial indexed field |

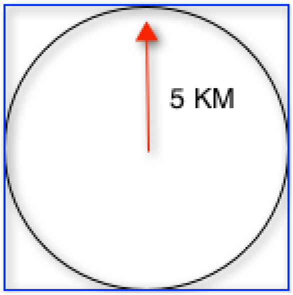
### `geofilt`

The `geofilt` filter allows you to retrieve results based on the geospatial distance (AKA the "great circle distance") from a given point. Another way of looking at it is that it creates a circular shape filter.  For example, to find all documents within five kilometers of a given lat/lon point, you could enter `&q=*:*&fq={!geofilt sfield=store}&pt=45.15,-93.85&d=5`. This filter returns all results within a circle of the given radius around the initial point:



### `bbox`

The `bbox` filter is very similar to `geofilt` except it uses the *bounding box* of the calculated circle. See the blue box in the diagram below. It takes the same parameters as geofilt. Here's a sample query: `&q=*:*&fq={!bbox sfield=store}&pt=45.15,-93.85&d=5`. The rectangular shape is faster to compute and so it's sometimes used as an alternative to geofilt when it's acceptable to return points outside of the radius. However, if the ideal goal is a circle but you want it to run faster, then instead consider using the RPT field and try a large "distErrPct" value like `0.1` (10% radius). This will return results outside the radius but it will do so somewhat uniformly around the shape.



> ⚠ When a bounding box includes a pole, the bounding box ends up being a "bounding bowl" (a *spherical cap*) that includes all values north of the lowest latitude of the circle if it touches the north pole (or south of the highest latitude if it touches the south pole).

#### Filtering by an arbitrary rectangle

Sometimes the spatial search requirement calls for finding everything in a rectangular area, such as the area covered by a map the user is looking at.  For this case, geofilt and bbox won't cut it.  This is somewhat of a trick, but you can use Solr's range query syntax for this by supplying the lower-left corner as the start of the range and the upper-right corner as the end of the range.  Here's an example:  `&q=*:*&fq=store:[45,-94 TO 46,-93]`.  LatLonType does **not** support rectangles that cross the dateline, but RPT does.  If you are using RPT with non-geospatial coordinates (`geo="false"`) then you must quote the points due to the space, e.g. `"x y"`.

#### Optimization: Solr Post Filtering

Most likely, the fastest spatial filters will be to simply use the RPT field type.  However, sometimes it may be faster to use LatLonType with *Solr post filtering* in circumstances when both the spatial query isn't worth caching and there aren't many matching documents that match the non-spatial filters (e.g. keyword queries and other filters).  To use *Solr post filtering* with LatLonType, use the `bbox` or `geofilt` query parsers in a filter query but specify `cache=false` and `cost=100` (or greater) as local-params. Here's a short example:

```
&q=...mykeywords...&fq=...someotherfilters...&fq={!geofilt cache=false
cost=100}&sfield=store&pt=45.15,-93.85&d=5
```

### Distance Function Queries

There are four distance function queries: `geodist`, see below, usually the most appropriate; `dist`, to calculate the p-norm distance between multi-dimensional vectors; `hsin`, to calculate the distance between two points on a sphere; and `sqedist`, to calculate the squared Euclidean distance between two points. For more information about these function queries, see the section on Function Queries.

### `geodist`

`geodist` is a distance function that takes three optional parameters: `(sfield,latitude,longitude)`. You can use the `geodist` function to sort results by distance or score return results.

For example, to sort your results by ascending distance, enter `...&q=*:*&fq={!geofilt}&sfield=store&pt=45.15,-93.85&d=50&sort=geodist asc`.

To return the distance as the document score, enter `...&q={!func}geodist()&sfield=store&pt=45.15,-93.85&sort=score+asc`.

## More Examples

Here are a few more useful examples of what you can do with spatial search in Solr.

### Use as a Sub-Query to Expand Search Results

Here we will query for results in Jacksonville, Florida, or within 50 kilometers of 45.15,-93.85 (near Buffalo, Minnesota):

```
&q=*:*&fq=(state:"FL" AND city:"Jacksonville") OR
{!geofilt}&sfield=store&pt=45.15,-93.85&d=50&sort=geodist()+asc
```

### Facet by Distance

To facet by distance, you can use the Frange query parser:

```
&q=*:*&sfield=store&pt=45.15,-93.85&facet.query={!frange l=0
u=5}geodist()&facet.query={!frange l=5.001 u=3000}geodist()
```

There are other ways to do it too, like using a {!geofilt} in each facet.query.

### Boost Nearest Results

Using the DisMax or Extended DisMax, you can combine spatial search with the boost function to boost the nearest results:

```
&q.alt=*:*&fq={!geofilt}&sfield=store&pt=45.15,-93.85&d=50&bf=recip(geodist(),2,200,20)&sort=score desc
```

## SpatialRecursivePrefixTreeFieldType (abbreviated as RPT)

Solr 4's new spatial field offers several new features and improvements over LatLonType:

- Query by polygons and other complex shapes, in addition to circles & rectangles
- Multi-valued indexed fields
- Ability to index non-point shapes (e.g. polygons) as well as point shapes
- Rectangles with user-specified corners that can cross the dateline
- Multi-value distance sort and score boosting *(warning: non-optimized)*
- Well-Known-Text (WKT) shape syntax (required for specifying polygons & other complex shapes)

RPT incorporates the basic features of LatLonType and PointType, such as lat-lon bounding boxes and circles. In fact you can (and should) use geofilt, bbox, geodist, and a range-query with it (which wasn't so when RPT was first introduced in Solr 4.0).

### Schema configuration

To use RPT, the field type must be registered and configured in `schema.xml`. There are many options for this field type.

| Setting | Description |
|---|---|
| name | The name of the field type. |
| class | This should be `solr.SpatialRecursivePrefixTreeFieldType`. But be aware that the Lucene spatial module includes some other so-called "spatial strategies" other than RPT, notably TermQueryPT*, BBox, PointVector*, and SerializedDV. Solr requires a field type to parallel these in order to use them. The asterisked ones have them. |
| spatialContextFactory | If polygons or linestrings are required, then JTS Topology Suite is a needed to implement them. It's a JAR file that you need to put on Solr's classpath (but not via the standard solrconfig.xml mechanisms). If you intend to use those shapes, set this attribute to `com.spatial4j.core.context.jts.JtsSpatialContextFactory`. Furthermore, the context factory has its own options which are directly configurable on the Solr field type here; follow the link to the Javadocs, and remember to look at the superclass's options in SpatialContextFactory as well. One option in particular you should most likely enable is a `utoIndex` (i.e. use PreparedGeometry) as it's been shown to be a major performance boost for polygons. Further details about specifying polygons to index or query are at Solr's Wiki linked below. |
| units | This is required, and currently can only be "degrees". It doesn't apply to geofilt, bbox, or geodist (which all use kilometers); it applies to maxDistErr and if you configure the query itself to return the distance. |
| distErrPct | Defines the default precision of non-point shapes (both index & query), as a fraction between 0.0 (fully precise) to 0.5. The closer this number is to zero, the more accurate the shape will be. However, more precise indexed shapes use more disk space and take longer to index. Bigger distErrPct values will make queries faster but less accurate. |
| maxDistErr | Defines the highest level of detail required for indexed data. If left blank, the default is one meter – just a bit less than 0.000009 degrees. This setting is used internally to compute an appropriate maxLevels (see below). |
| geo | If **true**, the default, latitude and longitude coordinates will be used and the mathematical model will generally be a sphere. If false, the coordinates will be generic X & Y on a 2D plane using Euclidean/Cartesian geometry. |
| worldBounds | Defines the valid numerical ranges for x and y, in the format of `ENVELOPE(minX, maxX, maxY, minY)`. If `geo="true"`, the standard lat-lon world boundaries are assumed. If `geo=false`, you should define your boundaries. |
| distCalculator | Defines the distance calculation algorithm. If `geo=true`, "haversine" is the default. If `geo=false`, "cartesian" will be the default. Other possible values are "lawOfCosines", "vincentySphere" and "cartesian^2". |
| prefixTree | Defines the spatial grid implementation. Since a PrefixTree (such as RecursivePrefixTree) maps the world as a grid, each grid cell is decomposed to another set of grid cells at the next level. If `geo=false` then the default prefix tree is "geohash", otherwise it's "quad". Geohash has 32 children at each level, quad has 4. Geohash cannot be used for `geo=false` as it's strictly geospatial. |

| | |
|---|---|
| maxLevels | Sets the maximum grid depth for indexed data. Instead, it's usually more intuitive to compute an appropriate maxLevels by specifying `maxDistErr` . |

```
<fieldType name="location_rpt"   class="solr.SpatialRecursivePrefixTreeFieldType"

spatialContextFactory="com.spatial4j.core.context.jts.JtsSpatialContextFactory"
               autoIndex="true"
               distErrPct="0.025"
               maxDistErr="0.000009"
               units="degrees" />
```

Once the field type has been defined, use it to define a field that uses it.

Because RPT has more advanced features, some of which are new and experimental, please review the Solr Wiki at http://wiki.apache.org/solr/SolrAdaptersForLuceneSpatial4 for more information about using this field type.

### BBoxField

The `BBoxField` field type indexes a single rectangle (bounding box) per document field and supports searching via a bounding box.  It supports most spatial search predicates, it has enhanced relevancy modes based on the overlap or area between the search rectangle and the indexed rectangle.  It's particularly useful for its relevancy modes.  To configure it in the schema, use a configuration like this:

```
<field name="bbox" type="bbox" />
<fieldType name="bbox" class="solr.BBoxField"
        geo="true" units="degrees" numberType="_bbox_coord" />
<fieldType name="_bbox_coord" class="solr.TrieDoubleField" precisionStep="8"
docValues="true" stored="false"/>
```

BBoxField is actually based off of 4 instances of another field type referred to by numberType.  It also uses a boolean to flag a dateline cross.  Assuming you want to use the relevancy feature, docValues is required.  Some of the attributes are in common with the RPT field like geo, units, worldBounds, and spatialContextFactory because they share some of the same spatial infrastructure.

To index a box, add a field value to a bbox field that's a string in the WKT/CQL ENVELOPE syntax.  Example: `ENVELOPE(-10, 20, 15, 10)` which is minX, maxX, maxY, minY order.  The parameter ordering is unintuitive but that's what the spec calls for.

To search, you can use the `{!bbox}` query parser, or the range syntax e.g. `[10,-10 TO 15,20]`, or the ENVELOPE syntax wrapped in parenthesis with a leading search predicate.  The latter is the only way to choose a predicate other than Intersects.  For example:

`&q={!field f=bbox}Contains(ENVELOPE(-10, 20, 15, 10))`

Now to sort the results by one of the relevancy modes, use it like this:

`&q={!field f=bbox score=overlapRatio}Intersects(ENVELOPE(-10, 20, 15, 10))`

The `score` local-param can be one of `overlapRatio`, `area`, and `area2D`.  area scores by the document area using surface-of-a-sphere (assuming geo=true) math, area2D uses simple width * height.  overlapRatio computes a [0-1] ranged score based on how much overlap exists relative to the document's area and the query area.  The javadocs of BBoxOverlapRatioValueSource have more info on the formula, if you're really curious.  There is an additional parameter queryTargetProportion that allows you to weight the query side of the formula to the index (target) side of the formula.  You can also use `&debug=results` to see useful score computation info.

## The Terms Component

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 search 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.

### Configuring the Terms Component

By default, the Terms Component is already configured in `solrconfig.xml` for each collection.

#### Defining the Terms Component

Defining the Terms search component is straightforward: simply give it a name and use the class `solr.TermsComponent`.

```
<searchComponent name="terms" class="solr.TermsComponent"/>
```

This makes the component available for use, but by itself will not be useable until included with a request handler.

#### Using the Terms Component in a Request Handler

The `/terms` request handler is also defined in `solrConfig.xml` by default.

```
<requestHandler name="/terms" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <bool name="terms">true</bool>
    <bool name="distrib">false</bool>
  </lst>
  <arr name="components">
    <str>terms</str>
  </arr>
</requestHandler>
```

Note that the defaults for the this request handler set the parameter "terms" to true, which allows terms to be returned on request. The parameter "distrib" is set to false, which allows this handler to be used only on a single Solr core. To finish out the configuration, he Terms Component is included as an available component to this request handler.

You could add this component to another handler if you wanted to, and pass "terms=true" in the HTTP request in order to get terms back. If it is only defined in a separate handler, you must use that handler when querying in order to get terms and not regular documents as results.

#### Terms Component Parameters

The parameters below allow you to control what terms are returned. You can also any of these to the request handler if you'd like to set them permanently. Or, you can add them to the query request. These parameters are:

| Parameter | Required | Default | Description |
|---|---|---|---|
| terms | No | false | If set to true, enables the Terms Component. By default, the Terms Component is off.<br>Example: `terms=true` |
| terms.fl | Yes | null | Specifies the field from which to retrieve terms.<br>Example: `terms.fl=title` |
| terms.limit | No | 10 | 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. Although this is not required, either this parameter or `terms.upper` must be defined.<br>Example: `terms.limit=20` |
| terms.lower | No | empty string | 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.<br>Example: `terms.lower=orange` |
| terms.lower.incl | No | true | If set to true, includes the lower-bound term (specified with `terms.lower` in the result set.<br>Example: `terms.lower.incl=false` |
| terms.mincount | No | null | 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 (that is, >= mincount).<br>Example: `terms.mincount=5` |
| terms.maxcount | No | null | 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 (that is, <= maxcount).<br>Example: `terms.maxcount=25` |
| terms.prefix | No | null | Restricts matches to terms that begin with the specified string.<br>Example: `terms.prefix=inter` |
| terms.raw | No | false | 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.<br>Example: `terms.raw=true` |

| | | | |
|---|---|---|---|
| terms.regex | No | null | Restricts matches to terms that match the regular expression.<br>Example: `terms.regex=*pedist` |
| terms.regex.flag | No | null | Defines a Java regex flag to use when evaluating the regular expression defined with `terms.regex`. See http://docs.oracle.com/javase/tutorial/essential/regex/pattern.html for details of each flag. Valid options are:<br>• case_insensitive<br>• comments<br>• multiline<br>• literal<br>• dotall<br>• unicode_case<br>• canon_eq<br>• unix_lines<br>Example: `terms.regex.flag=case_insensitive` |
| terms.sort | No | count | Defines how to sort the terms returned. Valid options are **count**, which sorts by the term frequency, with the highest term frequency first, or **index**, which sorts in index order.<br>Example: `terms.sort=index` |
| terms.upper | No | null | Specifies the term to stop at. Although this parameter is not required, either this parameter or `terms.limit` must be defined.<br>Example: `terms.upper=plum` |
| terms.upper.incl | No | false | If set to true, the upper bound term is included in the result set. The default is false.<br>Example: `terms.upper.incl=true` |

The output is a list of the terms and their document frequency values. See below for examples.

## Examples

The following examples use the sample Solr configuration located in the `<Solr>/example` directory and the sample documents in the `exampledocs` directory.

### Get Top 10 Terms

This query requests the first ten terms in the name field: `http://localhost:8983/solr/terms?terms.fl=name`

Results:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">2</int>
  </lst>
  <lst name="terms">
    <lst name="name">
      <int name="one">5</int>
      <int name="184">3</int>
      <int name="1gb">3</int>
      <int name="3200">3</int>
      <int name="400">3</int>
      <int name="ddr">3</int>
      <int name="gb">3</int>
      <int name="ipod">3</int>
      <int name="memory">3</int>
      <int name="pc">3</int>
    </lst>
  </lst>
</response>
```

#### Get First 10 Terms Starting with Letter 'a'

This query requests the first ten terms in the name field, in index order (instead of the top 10 results by document count): `http://localhost:8983/solr/terms?terms.fl=name&terms.lower=a&terms.sort=index`

Results:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
  </lst>
  <lst name="terms">
    <lst name="name">
      <int name="a">1</int>
      <int name="all">1</int>
      <int name="apple">1</int>
      <int name="asus">1</int>
      <int name="ata">1</int>
      <int name="ati">1</int>
      <int name="belkin">1</int>
      <int name="black">1</int>
      <int name="british">1</int>
      <int name="cable">1</int>
    </lst>
  </lst>
</response>
```

### Using the Terms Component for an Auto-Suggest Feature

If the Suggester doesn't suit your needs, 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:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <lst name="terms">
    <lst name="name">
      <int name="ata">1</int>
      <int name="ati">1</int>
    </lst>
  </lst>
</response>
```

You can use the parameter `omitHeader=true` to omit the response header from the query response, like in this example, which also returns the response in JSON format: `http://localhost:8983/solr/terms?terms.fl=name&terms.prefix=at&indent=true&wt=json&omitHeader=true`

Result:

```
{
  "terms": {
    "name": [
      "ata",
      1,
      "ati",
      1
    ]
  }
}
```

### Distributed Search Support

The TermsComponent also supports distributed indexes. For the `/terms` request handler, you must provide the following two parameters:

| Parameter | Description |
|---|---|
| shards | Specifies the shards in your distributed indexing configuration. For more information about distributed indexing, see Distributed Search with Index Sharding. |
| shards.qt | Specifies the request handler Solr uses for requests to shards. |

### More Resources

- TermsComponent wiki page
- TermsComponent javadoc

## The Term Vector Component

The TermVectorComponent is a search component designed to return additional information about documents matching your search.

For each document in the response, the TermVectorCcomponent can return the term vector, the term frequency, inverse document frequency, position, and offset information.

### Configuration

The TermVectorComponent is not enabled implicitly in Solr - it must be explicitly configured in your `solrconfig.xml` file.

To enable the this component, you need to configure it using a `searchComponent` element:

```
<searchComponent name="tvComponent"
class="org.apache.solr.handler.component.TermVectorComponent"/>
```

A request handler must then be configured to use this component name.  In this example, the component is associated with a special request handler named `/tvrh`, that enables term vectors by default using the `tv=true` parameter; but you can associate it with any request handler:

```
<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>
```

Once your handler is defined, you may use it to fetch term vectors for any fields configured with the `termVector` attribute in your `schema.xml`, for example:

```
<field name="includes"
       type="text"
       indexed="true"
       stored="true"
       multiValued="true"
       termVectors="true"
       termPositions="true"
       termOffsets="true" />
```

### Invoking the Term Vector Component

The example below shows an invocation of this component using the above configuration:

`http://localhost:8983/solr/collection1/tvrh?q=*%3A*&start=0&rows=10&fl=id,includes`

```
...
<lst name="termVectors">
   <lst name="warnings">
     <arr name="noTermVectors">
       <str>id</str>
     </arr>
   </lst>
   <lst name="doc-5">
     <str name="uniqueKey">MA147LL/A</str>
     <lst name="includes">
       <lst name="cabl"/>
       <lst name="earbud"/>
       <lst name="headphon"/>
       <lst name="usb"/>
     </lst>
   </lst>
   <str name="uniqueKeyFieldName">id</str>
   <lst name="doc-9">
     <str name="uniqueKey">3007WFP</str>
     <lst name="includes">
       <lst name="cabl"/>
       <lst name="usb"/>
     </lst>
   </lst>
   <str name="uniqueKeyFieldName">id</str>
   <lst name="doc-12">
     <str name="uniqueKey">9885A004</str>
     <lst name="includes">
       <lst name="32"/>
       <lst name="av"/>
       <lst name="batteri"/>
       <lst name="cabl"/>
       <lst name="card"/>
       <lst name="mb"/>
       <lst name="sd"/>
       <lst name="usb"/>
     </lst>
   </lst>
   <str name="uniqueKeyFieldName">id</str>
 </lst>
```

**Request Parameters**

The example below shows the available request parameters for this component:

`http://localhost:8983/solr/collection1/tvrh?q=*%3A*&version=2.2&start=0&rows=10&indent=on&qt=tvrh&tv=true&tv.tf=true&tv.df=true&tv.positions&tv.offsets=true`

| Boolean Parameters | Description | Type |
|---|---|---|
| tv | Should the component run or not | boolean |
| tv.docIds | Returns term vectors for the specified list of Lucene document IDs (not the Solr Unique Key). | comma seperated integers |

| | | |
|---|---|---|
| tv.fl | Returns term vectors for the specified list of fields. If not specified, the `fl` parameter is used. | comma seperated list of field names |
| tv.all | A shortcut that invokes all the boolean parameters listed below. | boolean |
| tv.df | Returns the Document Frequency (DF) of the term in the collection. This can be computationally expensive. | boolean |
| tv.offsets | Returns offset information for each term in the document. | boolean |
| tv.positions | Returns position information. | boolean |
| tv.tf | Returns document term frequency info per term in the document. | boolean |
| tv.tf_idf | Calculates TF*IDF for each term. Requires the parameters `tv.tf` and `tv.df` to be "true". This can be computationally expensive. (The results are not shown in example output) | boolean |

To learn more about TermVector component output, see the Wiki page: http://wiki.apache.org/solr/TermVectorComponentExampleOptions

For schema requirements, see the Wiki page: http://wiki.apache.org/solr/FieldOptionsByUseCase

### SolrJ and the Term Vector Component

Neither the SolrQuery class nor the QueryResponse class offer specific method calls to set Term Vector Component parameters or get the "termVectors" output. However, there is a patch for it: SOLR-949.

## The Stats Component

The Stats component returns simple statistics for numeric, string, and date fields within the document set.

### 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. |
| stats.calcdistinct | If **true**, distinct values will be calculated and returned as "countDistinct" and "distinctValues" in the response. This calculation may be expensive for some fields, so it is **false** by default. If you'd only like to return distinct values for specific fields, you can also specify `f.<field>.stats.calcdistinct`, replacing `<field>` with your field name, to limit the distinct value calculation to the required field. |

### 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. |
| distinctValues | Displays the distinct values in a field. |
| countDistinct | The number of distinct values in a field. |

### Example

The query below, which includes calculating distinct values would produce results like the ones shown below.

```
http://localhost:8983/solr/select?q=*:*&stats=true&stats.field=price&stats.field=
popularity&stats.calcdistinct=true&rows=0&indent=true
```

```
<lst name="stats">
   <lst name="stats_fields">
      <lst name="price">
         <double name="min">0.0</double>
         <double name="max">2199.0</double>
         <long name="count">16</long>
         <long name="missing">16</long>
         <arr name="distinctValues">
            <float>0.0</float>
            <float>11.5</float>
            <float>19.95</float>
            <float>74.99</float>
            <float>92.0</float>
            <float>179.99</float>
            <float>185.0</float>
            <float>279.95</float>
            <float>329.95</float>
            <float>350.0</float>
            <float>399.0</float>
            <float>479.95</float>
            <float>649.99</float>
            <float>2199.0</float>
         </arr>
         <long name="countDistinct">14</long>
         <double name="sum">5251.270030975342</double>
         <double name="sumOfSquares">6038619.175900028</double>
         <double name="mean">328.20437693595886</double>
         <double name="stddev">536.3536996709846</double>
         <lst name="facets" />
      </lst>
      <lst name="popularity">
         <double name="min">0.0</double>
         <double name="max">10.0</double>
         <long name="count">15</long>
         <long name="missing">17</long>
         <arr name="distinctValues">
            <int>0</int>
            <int>1</int>
            <int>5</int>
            <int>6</int>
            <int>7</int>
            <int>10</int>
         </arr>
         <long name="countDistinct">6</long>
         <double name="sum">85.0</double>
         <double name="sumOfSquares">603.0</double>
         <double name="mean">5.666666666666667</double>
         <double name="stddev">2.943920288775949</double>
         <lst name="facets" />
      </lst>
   </lst>
</lst>
```

Here are is a similar request with faceting requested for the field `inStock`, using the parameter `&stats.facet=inStock`. In this example, we have not requested distinct values to be calculated.

```
http://localhost:8983/solr/select?q=*:*&stats=true&stats.field=price&stats.field=
popularity&stats.facet=inStock&rows=0&indent=true
```

```
<lst name="stats">
   <lst name="stats_fields">
      <lst name="price">
         <double name="min">0.0</double>
         <double name="max">2199.0</double>
         <long name="count">16</long>
         <long name="missing">16</long>
         <double name="sum">5251.270030975342</double>
         <double name="sumOfSquares">6038619.175900028</double>
         <double name="mean">328.20437693595886</double>
         <double name="stddev">536.3536996709846</double>
         <lst name="facets">
            <lst name="inStock">
               <lst>
                  <double name="min">Infinity</double>
                  <double name="max">-Infinity</double>
                  <long name="count">0</long>
                  <long name="missing">11</long>
                  <double name="sum">0.0</double>
                  <double name="sumOfSquares">0.0</double>
                  <double name="mean">NaN</double>
                  <double name="stddev">0.0</double>
                  <lst name="facets" />
               </lst>
               <lst name="false">
                  <double name="min">11.5</double>
                  <double name="max">649.989990234375</double>
                  <long name="count">4</long>
                  <long name="missing">0</long>
                  <double name="sum">1161.3900032043457</double>
                  <double name="sumOfSquares">653369.2541528536</double>
                  <double name="mean">290.3475008010864</double>
                  <double name="stddev">324.63444532124953</double>
                  <lst name="facets" />
               </lst>
               <lst name="true">
                  <double name="min">0.0</double>
                  <double name="max">2199.0</double>
                  <long name="count">12</long>
                  <long name="missing">5</long>
                  <double name="sum">4089.880027770996</double>
                  <double name="sumOfSquares">5385249.921747174</double>
                  <double name="mean">340.823335647583</double>
                  <double name="stddev">602.3683083752779</double>
                  <lst name="facets" />
               </lst>
            </lst>
         </lst>
      </lst>
      <lst name="popularity">
         <double name="min">0.0</double>
         <double name="max">10.0</double>
         <long name="count">15</long>
         <long name="missing">17</long>
         <double name="sum">85.0</double>
         <double name="sumOfSquares">603.0</double>
         <double name="mean">5.666666666666667</double>
         <double name="stddev">2.943920288775949</double>
         <lst name="facets">
            <lst name="inStock">
               <lst>
                  <double name="min">Infinity</double>
                  <double name="max">-Infinity</double>
                  <long name="count">0</long>
                  <long name="missing">11</long>
                  <double name="sum">0.0</double>
                  <double name="sumOfSquares">0.0</double>
                  <double name="mean">NaN</double>
                  <double name="stddev">0.0</double>
                  <lst name="facets" />
               </lst>
               <lst name="false">
                  <double name="min">1.0</double>
                  <double name="max">7.0</double>
                  <long name="count">4</long>
                  <long name="missing">0</long>
                  <double name="sum">16.0</double>
                  <double name="sumOfSquares">100.0</double>
                  <double name="mean">4.0</double>
                  <double name="stddev">3.4641016151377544</double>
                  <lst name="facets" />
               </lst>
               <lst name="true">
                  <double name="min">0.0</double>
                  <double name="max">10.0</double>
                  <long name="count">11</long>
                  <long name="missing">6</long>
                  <double name="sum">69.0</double>
                  <double name="sumOfSquares">503.0</double>
                  <double name="mean">6.2727272727272725</double>
                  <double name="stddev">2.6491851234260353</double>
                  <lst name="facets" />
               </lst>
            </lst>
         </lst>
      </lst>
```

```
  </lst>
</lst>
```

### Local Parameters

Similar to the Facet Component, the `stats.field` parameter supports local parameters for:

- Tagging & Excluding Filters: `stats.field={!ex=filterA}price`
- Changing the Output Key: `stats.field={!key=my_price_stats}price`

#### Example

Here we compute stats for the price field - once including the filter on the inStock field, and once excluding it:

```
http://localhost:8983/solr/select?q=*:*&fq={!tag=stock_check}inStock:true&stats=true&stats.field={!ex=stock_check+key=instock_prices}price&stats.field={!key=all_prices}price&rows=0&indent=true
```

```
<lst name="stats">
  <lst name="stats_fields">
    <lst name="instock_prices">
      <double name="min">0.0</double>
      <double name="max">2199.0</double>
      <long name="count">16</long>
      <long name="missing">16</long>
      <double name="sum">5251.270030975342</double>
      <double name="sumOfSquares">6038619.175900028</double>
      <double name="mean">328.20437693595886</double>
      <double name="stddev">536.3536996709846</double>
      <lst name="facets"/>
    </lst>
    <lst name="all_prices">
      <double name="min">0.0</double>
      <double name="max">2199.0</double>
      <long name="count">12</long>
      <long name="missing">5</long>
      <double name="sum">4089.880027770996</double>
      <double name="sumOfSquares">5385249.921747174</double>
      <double name="mean">340.823335647583</double>
      <double name="stddev">602.3683083752779</double>
      <lst name="facets"/>
    </lst>
  </lst>
</lst>
```

### 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
```

> All facet results are returned, so 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.

## The Query Elevation Component

The Query Elevation Component lets you configure the top results for a given query regardless of the normal Lucene scoring. This is sometimes called "sponsored search," "editorial boosting," or "best bets." This component matches the user query text to a configured map of top results. The text can be any string or non-string IDs, as long as it's indexed. Although this component will work with any QueryParser, it makes the most sense to use with DisMax or eDisMax.

The Query Elevation Component is supported by distributed searching.

### Configuring the Query Elevation Component

You can configure the Query Elevation Component in the `solrconfig.xml` file. The default configuration looks like this:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
  <!-- pick a fieldType to analyze queries -->
  <str name="queryFieldType">string</str>
  <str name="config-file">elevate.xml</str>
</searchComponent>

<requestHandler name="/elevate" class="solr.SearchHandler" startup="lazy">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
  </lst>
  <arr name="last-components">
    <str>elevator</str>
  </arr>
</requestHandler>
```

Optionally, in the Query Elevation Component configuration you can also specify the following to distinguish editorial results from "normal" results:

```
<str name="editorialMarkerFieldName">foo</str>
```

The Query Elevation Search Component takes the following arguments:

| Argument | Description |
|---|---|
| `queryFieldType` | Specifies which fieldType should be used to analyze the incoming text. For example, it may be appropriate to use a fieldType with a LowerCaseFilter. |
| `config-file` | Path to the file that defines query elevation. This file must exist in `<instanceDir>/conf/<config-file>` or `<dataDir>/<config-file>`.<br>If the file exists in the /conf/ directory it will be loaded once at startup. If it exists in the data directory, it will be reloaded for each IndexReader. |
| `forceElevation` | By default, this component respects the requested `sort` parameter: if the request asks to sort by date, it will order the results by date. If `forceElevation=true` (the default), results will first return the boosted docs, then order by date. |

### `elevate.xml`

Elevated query results are configured in an external XML file specified in the `config-file` argument. An `elevate.xml` file might look like this:

```
<elevate>
 <query text="AAA">
  <doc id="A" />
  <doc id="B" />
 </query>

 <query text="ipod">
  <doc id="A" />

  <!-- you can optionally exclude documents from a query result -->

  <doc id="B" exclude="true" />
 </query>
</elevate>
```

In this example, the query "AAA" would first return documents A and B, then whatever normally appears for the same query. For the query "ipod", it would first return A, and would make sure that B is not in the result set.

## Using the Query Elevation Component

### The `enableElevation` Parameter

For debugging it may be useful to see results with and without the elevated docs. To hide results, use `enableElevation=false`:

`http://localhost:8983/solr/elevate?q=YYYY&debugQuery=true&enableElevation=true`

`http://localhost:8983/solr/elevate?q=YYYY&debugQuery=true&enableElevation=false`

### The `forceElevation` Parameter

You can force elevation during runtime by adding `forceElevation=true` to the query URL:

`http://localhost:8983/solr/elevate?q=YYYY&debugQuery=true&enableElevation=true&forceElevation=true`

### The `exclusive` Parameter

You can force Solr to return only the results specified in the elevation file by adding `exclusive=true` to the URL:

`http://localhost:8983/solr/elevate?q=YYYY&debugQuery=true&exclusive=true`

### Document Transformers and the `markExcludes` Parameter

The `[elevated]` Document Transformer can be used to annotate each document with information about whether or not it was elevated:

`http://localhost:8983/solr/elevate?q=YYYY&fl=id,[elevated]`

Likewise, it can be helpful when troubleshooting to see all matching documents – including documents that the elevation configuration would normally exclude.  This is possible by using the `markExcludes=true` parameter, and then using the `[excluded]` transformer:

`http://localhost:8983/solr/elevate?q=YYYY&markExcludes=true&fl=id,[elevated],[excluded]`

#### The `elevateIds` and `excludeIds` Parameters

When the elevation component is in use, the pre-configured list of elevations for a query can be overridden at request time to use the unique keys specified in these request parameters.

For example, in the request below documents A and B will be elevated, and document C will be excluded -- regardless of what elevations or exclusions are configured for the query YYYY in elevate.xml:

`http://localhost:8983/solr/elevate?q=YYYY&excludeIds=C&elevateIds=A,B`

If either one of these parameters is specified at request time, the the entire elevation configuration for the query is ignored.

For example, in the request below documents A and B will be elevated, and no documents will be excluded – regardless of what elevations or exclusions are configured for the query YYYY in elevate.xml:

`http://localhost:8983/solr/elevate?q=YYYY&elevateIds=A,B`

#### The `fq` Parameter

Query elevation respects the standard filter query (`fq`) parameter. That is, if the query contains the `fq` parameter, all results will be within that filter even if `elevate.xml` adds other documents to the result set.

## 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 |
|---|---|
| csv | CSVResponseWriter |
| json | JSONResponseWriter |
| php | PHPResponseWriter |
| phps | PHPSerializedResponseWriter |
| python | PythonResponseWriter |
| ruby | RubyResponseWriter |
| velocity | VelocityResponseWriter |
| xml | XMLResponseWriter |
| xslt | XSLTResponseWriter |

### 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 XSLT Response Writer 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.

### 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 when the Solr server is upgraded.

| XML Version | Notes | Comments |
|---|---|---|
| 2.0 | An `<arr>` tag was used for multiValued fields only if there was more then one value. | Not supported in Solr 4. |
| 2.1 | An `<arr>` tag is used for multiValued fields even if there is only one value. | Not supported in Solr 4. |
| 2.2 | The format of the responseHeader changed to use the same `<lst>` structure as the rest of the response. | Supported in Solr 4. |

The default value is the latest supported.

### 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.

> ⚠ 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.

### 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.

## 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.

### `tr` Parameter

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"/>`

#### 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.

### JSON Response Writer

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.

With Solr 4, the `JsonResponseWriter` has been changed:

- The default mime type for the writer is now `application/json`.
- The example solrconfig.xml has been updated to explicitly use this parameter to set the type to `text/plain`:

```
<queryResponseWriter name="json" class="solr.JSONResponseWriter">
  <!-- For the purposes of the tutorial, JSON response are written as
       plain text so that it's easy to read in *any* browser.
       If you are building applications that consume JSON, just remove
       this override to get the default "application/json" mime type.
    -->
  <str name="content-type">text/plain</str>
</queryResponseWriter>
```

### Python Response Writer

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 the python 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

### PHP Response Writer and PHP Serialized Response Writer

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 PHP Serialized Response Writer 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=*php{*}s');
$result = unserialize($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"/>
```

### Ruby Response Writer

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'\] }
```

### CSV Response Writer

The CSV response writer returns a list of documents in comma-separated values (CSV) format. Other information that would normally be included in a response, such as facet information, is excluded.

The CSV response writer supports multi-valued fields, and the output of this CSV format is compatible with Solr's CS

V update format. As of Solr 4.3, it can also support pseudo-fields.

#### CSV Parameters

These parameters specify the CSV format that will be returned. You can accept the default values or specify your own.

| Parameter | Default Value |
|---|---|
| csv.encapsulator | " |
| csv.escape | None |
| csv.separator | , |
| csv.header | Defaults to true. If false, Solr does not print the column headers |
| csv.newline | \n |
| csv.null | Defaults to a zero length string. Use this parameter when a document has no value for a particular field. |

#### Multi-Valued Field CSV Parameters

These parameters specify how multi-valued fields are encoded. Per-field overrides for these values can be done using `f.<fieldname>.csv.separator=|`.

| Parameter | Default Value |
|---|---|
| csv.mv.encapsulator | None |
| csv.mv.escape | \ |
| csv.mv.separator | Defaults to the `csv.separator` value |

#### Example

`http://localhost:8983/solr/select?q=ipod&fl=id,cat,name,popularity,price,score&wt=csv` returns:

```
id,cat,name,popularity,price,score
IW-02,"electronics,connector",iPod & iPod Mini USB 2.0 Cable,1,11.5,0.98867977
F8V7067-APL-KIT,"electronics,connector",Belkin Mobile Power Cord for iPod w/
Dock,1,19.95,0.6523595
MA147LL/A,"electronics,music",Apple 60 GB iPod with Video Playback
Black,10,399.0,0.2446348
```

### Velocity Response Writer

The VelocityResponseWriter (also known as Solritas) is an optional plugin available in the `contrib/velocity` directory. It is used to power the Velocity Search UI in the example configuration.

Its jar and dependencies must be added (via <lib> or solr/home lib inclusion), and must be registered in solrconfig.xml like this:

```
<queryResponseWriter name="velocity" class="solr.VelocityResponseWriter"/>
```

For more information about the Velocity Response Writer, see https://wiki.apache.org/solr/VelocityResponseWriter.

### Binary Response Writer

Solr also includes a Response Writer that outputs binary format for use with a Java client. See Client APIs for more details.

## Near Real Time Searching

Near Real Time (NRT) search means that documents are available for search almost immediately after being indexed: additions and updates to documents are seen in 'near' real time. Solr 4 no longer blocks updates while a commit is in progress. Nor does it wait for background merges to complete before opening a new search of indexes and returning.

With NRT, you can modify a `commit` command to be a **soft commit**, which avoids parts of a standard commit that can be costly. You will still want to do standard commits to ensure that documents are in stable storage, but **soft commits** let you see a very near real time view of the index in the meantime. However, pay special attention to cache and autowarm settings as they can have a significant impact on NRT performance.

### Commits and Optimizing

A commit operation makes index changes visible to new search requests. A **hard commit** uses the transaction log to get the id of the latest document changes, and also calls `fsync` on the index files to ensure they have been flushed to stable storage and no data loss will result from a power failure.

A **soft commit** is much faster since it only makes index changes visible and does not `fsync` index files or write a new index descriptor. If the JVM crashes or there is a loss of power, changes that occurred after the last **hard commit** will be lost. Search collections that have NRT requirements (that want index changes to be quickly visible to searches) will want to soft commit often but hard commit less frequently. A softCommit may be "less expensive" in terms of time, but not free, since it can slow throughput.

An **optimize** is like a **hard commit** except that it forces all of the index segments to be merged into a single segment first. Depending on the use, this operation should be performed infrequently (e.g., nightly), if at all, since it involves reading and re-writing the entire index. Segments are normally merged over time anyway (as determined by the merge policy), and optimize just forces these merges to occur immediately.

Soft commit takes uses two parameters: `maxDocs` and `maxTime`.

| Parameter | Description |
|---|---|
| `maxDocs` | Integer. Defines the number of documents to queue before pushing them to the index. It works in conjunction with the `update_handler_autosoftcommit_max_time` parameter in that if either limit is reached, the documents will be pushed to the index. |
| `maxTime` | The number of milliseconds to wait before pushing documents to the index. It works in conjunction with the `update_handler_autosoftcommit_max_docs` parameter in that if either limit is reached, the documents will be pushed to the index. |

Use `maxDocs` and `maxTime` judiciously to fine-tune your commit strategies.

#### AutoCommits

An autocommit also uses the parameters `maxDocs` and `maxTime`. However it's useful in many strategies to use both a hard `autocommit` and `autosoftcommit` to achieve more flexible commits.

A common configuration is to do a hard `autocommit` every 1-10 minutes and a `autosoftcommit` every second. With this configuration, new documents will show up within about a second of being added, and if the power goes out, soft commits are lost unless a hard commit has been done.

For example:

```
<autoSoftCommit>
  <maxTime>1000</maxTime>
</autoSoftCommit>
```

It's better to use `maxTime` rather than `maxDocs` to modify an `autoSoftCommit`, especially when indexing a large number of documents through the commit operation. It's also better to turn off `autoSoftCommit` for bulk indexing.

#### Optional Attributes for `commit` and `optimize`

| Parameter | Valid Attributes | Description |
|---|---|---|
| `waitSearcher` | true, false | Block until a new searcher is opened and registered as the main query searcher, making the changes visible. Default is true. |
| `softCommit` | true, false | Perform a soft commit. This will refresh the view of the index faster, but without guarantees that the document is stably stored. Default is false. |
| `expungeDeletes` | true, false | Valid for `commit` only. This parameter purges deleted data from segments. The default is false. |
| `maxSegments = N` | integer | Valid for `optimize` only. Optimize down to at most this number of segments. The default is 1. |

Example of `commit` and `optimize` with optional attributes:

```
<commit waitSearcher="false"/>
<commit waitSearcher="false" expungeDeletes="true"/>
<optimize waitSearcher="false"/>
```

#### Passing `commit` and `commitWithin` parameters as part of the URL

Update handlers can also get `commit`-related parameters as part of the update URL. This example adds a small test document and causes an explicit commit to happen immediately afterwards:

```
http://localhost:8983/solr/update?stream.body=<add><doc>
   <field name="id">testdoc</field></doc></add>&commit=true
```

Alternately, you may want to use this:

```
http://localhost:8983/solr/update?stream.body=<optimize/>
```

This example causes the index to be optimized down to at most 10 segments, but won't wait around until it's done (`w` aitFlush=false):

```
curl 'http://localhost:8983/solr/update?optimize=true&maxSegments=10&waitFlush=false'
```

This example adds a small test document with a commitWithin instruction that tells Solr to make sure the document is committed no later than 10 seconds later (this method is generally preferred over explicit commits):

```
curl http://localhost:8983/solr/update?commitWithin=10000
  -H "Content-Type: text/xml" --data-binary
  '<add><doc><field name="id">testdoc</field></doc></add>'
```

#### Changing default commitWithin Behavior

The commitWithin settings allow forcing document commits to happen in a defined time period. This is used most frequently with Near Real Time Searching, and for that reason the default is to perform a soft commit. This does not, however, replicate new documents to slave servers in a master/slave environment. If that's a requirement for your implementation, you can force a hard commit by adding a parameter, as in this example:

```
<commitWithin>
  <softCommit>false</softCommit>
</commitWithin>
```

With this configuration, when you call commitWithin as part of your update message, it will automatically perform a hard commit every time.

## RealTime Get

For index updates to be visible (searchable), some kind of commit must reopen a searcher to a new point-in-time view of the index. The **realtime get** feature allows retrieval (by unique-key) of the latest version of any documents without the associated cost of reopening a searcher. This is primarily useful when using Solr as a NoSQL data store and not just a search index.

Realtime Get currently relies on the update log feature, which is enabled by default. It relies on an update log, which is configured in solrconfig.xml, in a section like:

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
</updateLog>
```

The latest example solrconfig.xml should also have a request handler named /get already defined like the following:

```
<requestHandler name="/get" class="solr.RealTimeGetHandler">
  <lst name="defaults">
    <str name="omitHeader">true</str>
    <str name="wt">json</str>
    <str name="indent">true</str>
  </lst>
</requestHandler>
```

Start (or restart) the Solr server, and then index a document:

```
curl 'http://localhost:8983/solr/update/json?commitWithin=10000000'
  -H 'Content-type:application/json' -d '[{"id":"mydoc","title":"realtime-get
test!"}]'
```

If you do a normal search, this document should not be found:

```
http://localhost:8983/solr/select?q=id:mydoc
...
"response":
{"numFound":0,"start":0,"docs":[]}
```

However if you use the realtime get handler exposed at `/get`, you should be able to retrieve that document:

```
http://localhost:8983/solr/get?id=mydoc
...
{"doc":{"id":"mydoc","title":"realtime-get test!"]}}
```

You can also specify multiple documents at once via the **ids** parameter and a comma separated list of ids, or by using multiple **id** parameters. If you specify multiple ids, or use the **ids** parameter, the response will mimic a normal query response to make it easier for existing clients to parse. Since you've only indexed one document, the following equivalent examples just repeat the same id.

```
http://localhost:8983/solr/get?ids=mydoc,mydoc
http://localhost:8983/solr/get?id=mydoc&id=mydoc
...
{"response":
  {"numFound":2,"start":0,"docs":
    [ { "id":"mydoc", "title":["realtime-get test!"]},
      { "id":"mydoc", "title":["realtime-get test!"]}]
  }
}
```

Do **NOT** disable the realtime get handler at `/get` if you are using SolrCloud otherwise any leader election will cause a full sync in **ALL** replicas for the shard in question. Similarly, a replica recovery will also always fetch the complete index from the leader because a partial sync will not be possible in the absence of this handler.

## Exporting Result Sets

Starting with Solr 4.10, it's possible to allow users to export fully sorted result sets. It is specifically designed to handle scenarios that involve sorting and exporting millions of records. It uses a stream sorting technique that begins to send records fairly within milliseconds and continues to stream results until the entire result set has been sorted and exported.

The cases where this functionality may be useful include: session analysis, distributed merge joins, time series roll-ups, aggregations on high cardinality fields, fully distributed field collapsing, and sort based stats.

### Field Requirements

All the fields being sorted and exported must have docValues set to true. For more information, see the section on DocValues.

You can choose between the different docValues formats to trade off memory usage and performance. The fastest is likely to be the “Direct” doc values format as it is uncompressed and fully in-memory. The initial tests were performed with the default Lucene410 docValues format and the “Direct” doc values format.

### Defining the /export Request Handler

To export the full sorted result set you use the new /export request handler.

This request handler is included in the example `solrconfig.xml` and if you use that as the basis for your own new Solr implementation you already have it configured. If however, you would like to add to your existing `solrconfig.xml`, you can add a section like this:

```
<requestHandler name="/export" class="solr.SearchHandler">
    <lst name="invariants">
      <str name="rq">{!xport}</str>
      <str name="wt">xsort</str>
      <str name="distrib">false</str>
    </lst>
    <arr name="components">
      <str>query</str>
    </arr>
  </requestHandler>
```

Note that this request handler's properties are defined as "invariants", which means they cannot be overridden by other properties passed at another time (such as at query time).

### Requesting Results Export

Once the /export request handler is defined, you can use it to make requests to export the result set of a query.

All queries must include `sort` and `fl` parameters, or the query will return an error. Filter queries are also supported. Results are always returned in JSON format.

The basic syntax is as follows:

```
http://localhost:8983/solr/export?q=my-query&sort=fieldA desc,fieldB
desc&fl=fieldA,fieldB,fieldC
```

#### Specifying the Sort Criteria

The `sort` property defines how documents will be sorted in the exported result set. Results can be sorted by any field that has a field type of int,long, float, double, string. The sort fields must be single valued fields.

Up to four sort fields can be specified per request, with the 'asc' or 'desc' properties.

#### Specifying the Field List

The `fl` property defines the fields that will be exported with the result set. Any of the field types that can be sorted (i.e., int, long, float, double, string) can be used in the field list. The fields can be single or multi-valued. However, returning scores and wildcards are not supported at this time.

### Distributed Support

The initial release treats all queries as non-distributed requests. So the client is responsible for making the calls to each Solr instance and merging the results.

Using SolrJ’s CloudSolrServer as a model, developers could build clients that automatically send requests to all the shards in a collection (or multiple collections) and then merge the sorted sets any way they wish.

# The Well-Configured Solr Instance

This section tells you how to fine-tune your Solr instance for optimum performance. This section covers the following topics:

Configuring solrconfig.xml: Describes how to work with the main configuration file for Solr, `solrconfig.xml`, covering the major sections of the file.

Solr Cores and solr.xml: Describes how to work with `solr.xml` and `core.properties` to configure your Solr core, or multiple Solr cores within a single instance.

Solr Plugins: Introduces Solr plugins with pointers to more information.

JVM Settings: Gives some guidance on best practices for working with Java Virtual Machines.

> ⚠ The focus of this section is generally on configuring a single Solr instance, but for those interested in scaling a Solr implementation in a cluster environment, see also the section SolrCloud. There are also options to scale through sharding or replication, described in the section Legacy Scaling and Distribution.

## Configuring solrconfig.xml

The `solrconfig.xml` file is the configuration file with the most parameters affecting Solr itself. While configuring Solr, you'll work with `solrconfig.xml` often. 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 Legacy Scaling and Distribution)

The `solrconfig.xml` file is found in the `solr/conf/` directory. The example file is well-commented, and includes information on best practices for most installations.

We've covered the options in the following sections:

- DataDir and DirectoryFactory in SolrConfig
- Lib Directives in SolrConfig
- Managed Schema Definition in SolrConfig
- IndexConfig in SolrConfig
- UpdateHandlers in SolrConfig
- Query Settings in SolrConfig
- RequestDispatcher in SolrConfig
- RequestHandlers and SearchComponents in SolrConfig

### Substituting Properties in Solr Config Files

Solr supports variable substitution of property values in config files, which allows runtime specification of various configuration options in `solrconfig.xml`. The syntax is `${propertyname[:option default value]}`. This allows defining a default that can be overridden when Solr is launched. If a default value is not specified, then the property *must* be specified at runtime or the configuration file will generate an error when parsed.

There are multiple methods for specifying properties that can be used in configuration files.

### JVM System Properties

Any JVM System properties, usually specified using the `-D` flag when starting the JVM, can be used as variables in any XML configuration file in Solr.

For example, in the example `solrconfig.xml`, you will see this value which defines the locking type to use:

```
<lockType>${solr.lock.type:native}</lockType>
```

Which means the lock type defaults to "native" but when starting Solr's example application, you could override this by launching the JVM it with:

```
java -Dsolr.lock.type=simple -jar start.jar
```

### `solrcore.properties`

If the configuration directory for a Solr core contains a file named `solrcore.properties` that file can contain any arbitrary user defined property names and values using the Java standard properties file format, and those properties can be used as variables in the XML configuration files for that Solr core.

For example, the following `solrcore.properties` file could be created in the `solr/collection1/conf` directory of the Solr example configuration, to specify the lockType used.

```
#conf/solrcore.properties
lock.type=simple
```

> The path and name of the `solrcore.properties` file can be overridden using the `properties` property in `core.properties`

### User defined properties from `core.properties`

If you are using the newer core discovery style solr.xml such that each Solr core has a `core.properties` file, then any *user defined* properties in that file may be specified there and those properties will be available for substitution when parsing XML configuration files for that Solr core.

For example, consider the following `core.properties` file:

```
#core.properties
name=collection2
my.custom.prop=edismax
```

the `my.custom.prop` property can be used as a variable, like so...

```
<requestHandler name="/select">
    <lst name="defaults">
      <str name="defType">${my.custom.prop}</str>
    </lst>
  </requestHandler>
```

#### User defined properties from the Legacy `solr.xml` Format

Similar to the `core.properties` option above, user defined properties may be specified in the legacy solr.xml format. Please see the "User Defined Properties in solr.xml" section of the Legacy solr.xml Configuration documentation for more details.

#### Implicit Core Properties

Several attributes of a Solr core are available as "implicit" properties that can be used in variable substitution, independent of where or how they underlying value is initialized. For example: regardless of whether the name for a particular Solr core is explicitly configured in `core.properties` or inferred from the name of the instance directory, the implicit property `solr.core.name` is available for use as a variable in that core's configuration file...

```
<requestHandler name="/select">
    <lst name="defaults">
      <str name="collection_name">${solr.core.name}</str>
    </lst>
  </requestHandler>
```

All implicit properties use the `solr.core.` name prefix, and reflect the runtime value of the equivalent `core.properties` property:

- `solr.core.name`
- `solr.core.config`
- `solr.core.schema`
- `solr.core.dataDir`
- `solr.core.transient`
- `solr.core.loadOnStartup`

### More Information

- The Solr Wiki has a comprehensive page on `solrconfig.xml`, at http://wiki.apache.org/solr/SolrConfigXml.
- 6 Sins of solrconfig.xml modifications from solr.pl.

## DataDir and DirectoryFactory in SolrConfig

#### 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 Legacy Scaling and Distribution), then the <da

`taDir>` directory should correspond to the index directory used in the replication configuration.

#### Specifying the DirectoryFactory For Your Index

The default `solr.StandardDirectoryFactory` is filesystem based, and tries to pick the best implementation for the current JVM and platform. You can force a particular implementation by specifying `solr.MMapDirectoryFactory`, `solr.NIOFSDirectoryFactory`, or `solr.SimpleFSDirectoryFactory`.

```
<directoryFactory name="DirectoryFactory"
                  class="${solr.directoryFactory:solr.StandardDirectoryFactory}"/>
```

The `solr.RAMDirectoryFactory` is memory based, not persistent, and does not work with replication. Use this DirectoryFactory to store your index in RAM.

```
<directoryFactory class="org.apache.solr.core.RAMDirectoryFactory"/>
```

## Lib Directives in SolrConfig

Solr allows loading plugins by defining `<lib/>` directives in `solrconfig.xml`.

The plugins are loaded in the order they appear in `solrconfig.xml`. If there are dependencies, list the lowest level dependency jar first.

Regular expressions can be used to provide control loading jars with dependencies on other jars in the same directory. All directories are resolved as relative to the Solr `instanceDir`.

```
<lib dir="../../../contrib/extraction/lib" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-cell-\d.*\.jar" />

<lib dir="../../../contrib/clustering/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-clustering-\d.*\.jar" />

<lib dir="../../../contrib/langid/lib/" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-langid-\d.*\.jar" />

<lib dir="../../../contrib/velocity/lib" regex=".*\.jar" />
<lib dir="../../../dist/" regex="solr-velocity-\d.*\.jar" />
```

## Managed Schema Definition in SolrConfig

The Schema API enables schema modifications through a REST interface. (Read-only access to all schema elements is also supported.)

There are challenges with allowing programmatic access to a configuration file that is also open to manual edits: system-generated and manual edits may overlap and the system-generated edits may remove comments or other customizations that are critical for the organization to understand why fields, field types, etc., are defined the way they are. You may want to version the file with source control, or limit manual edits altogether.

`solrconfig.xml` allows the Solr schema to be defined as a "managed index schema": schema modification is only possible through the Schema API.

From the example `solrconfig.xml`:

```
<!-- To enable dynamic schema REST APIs, use the following for <schemaFactory>:

       <schemaFactory class="ManagedIndexSchemaFactory">
         <bool name="mutable">true</bool>
         <str name="managedSchemaResourceName">managed-schema</str>
       </schemaFactory>

       When ManagedIndexSchemaFactory is specified, Solr will load the schema from
       the resource named in 'managedSchemaResourceName', rather than from schema.xml.
       Note that the managed schema resource CANNOT be named schema.xml.  If the
managed
       schema does not exist, Solr will create it after reading schema.xml, then
rename
       'schema.xml' to 'schema.xml.bak'.

       Do NOT hand edit the managed schema - external modifications will be ignored
and
       overwritten as a result of schema modification REST API calls.

       When ManagedIndexSchemaFactory is specified with mutable = true, schema
       modification REST API calls will be allowed; otherwise, error responses will be
       sent back for these requests.
  -->

  <schemaFactory class="ClassicIndexSchemaFactory"/>
```

In the example above, `solrconfig.xml` is actually configured to use the `ClassicIndexSchemaFactory`, which treats the `schema.xml` file the same as it always has, which is that it can be edited manually. This setting disallows Schema API methods that modify the schema.

In the commented out sample, however, you can see configuration for the managed schema. In order for schema modifications to be possible via the Schema API, the `ManagedIndexSchemaFactory` will need to be used. The parameter `mutable` must also be set to **true**. The `managedSchemaResourceName`, which defaults to "managed-schema", may also be defined, and can be anything other than "schema.xml". Once Solr is restarted, the existing `schema.xml` file is renamed to `schema.xml.bak` and the contents are written to a file with the name defined as the `managedSchemaResourceName`. If you look at the resulting file, you'll see this at the top of the page:

```
<!-- Solr managed schema - automatically generated - DO NOT EDIT -->
```

Note that the Schemaless Mode example at `example/example-schemaless/` uses the `ManagedIndexSchemaFactory` to allow automatic schema field additions based on document updates' field values.

## IndexConfig in SolrConfig

The `<indexConfig>` section of `solrconfig.xml` defines low-level behavior of the Lucene index writers. By default, the settings are commented out in the sample `solrconfig.xml` included with Solr, which means the defaults are used. In most cases, the defaults are fine.

```
<indexConfig>
  ...
</indexConfig>
```

> Prior to Solr 4, many of these settings were contained in sections called `mainIndex` and `indexDefaults`. In Solr 4, those sections are deprecated and removed. Any settings that used to be in those sections, now belong in `<indexConfig>`.

Parameters covered in this section:

## Sizing Index Segments

### ***`ramBufferSizeMB`***

Once accumulated document updates exceed this much memory space (defined 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. The default is 100Mb (raised from 32Mb for Solr 4.1).

```
<ramBufferSizeMB>100</ramBufferSizeMB>
```

### ***`maxBufferedDocs`***

Sets the number of document updates to buffer in memory before they are flushed as a new segment. This may also trigger a merge. The default Solr configuration sets to flush by RAM usage (`ramBufferSizeMB`).

```
<maxBufferedDocs>1000</maxBufferedDocs>
```

### ***`maxIndexingThreads`***

The maximum number of simultaneous threads used to index documents. Once this threshold is reached, additional threads will wait for the others to finish. The default is 8. This parameter is new for Solr 4.1.

```
<maxIndexingThreads>8</maxIndexingThreads>
```

### ***`UseCompoundFile`***

Setting `<useCompoundFile>` to **true** combines the various files of a segment into a single file, although the default is **false**. On systems where the number of open files allowed per process is limited, setting this to **false** 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). In some cases, other internal factors may set a segment to "compound=false", even if this is setting is explicitly set to true, so the compounding of the files in a segment may not always happen.

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.

The default is **false**.

```
<useCompoundFile>false</useCompoundFile>
```

### Merging Index Segments

#### *`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.

```
<mergeFactor>10</mergeFactor>
```

#### *`mergePolicy`*

Defines how merging segments is done. The default in Solr is `TieredMergePolicy`, which merges segments of approximately equal size, subject to an allowed number of segments per tier. Other policies available are the `LogByteSizeMergePolicy` and `LogDocMergePolicy`. For more information on these policies, please see the MergePolicy javadocs.

```
<mergePolicy class="org.apache.lucene.index.TieredMergePolicy">
  <int name="maxMergeAtOnce">10</int>
  <int name="segmentsPerTier">10</int>
</mergePolicy>
```

#### *mergeScheduler*

The merge scheduler controls how merges are performed. The default `ConcurrentMergeScheduler` performs merges in the background using separate threads. The alternative, `SerialMergeScheduler`, does not perform merges with separate threads.

```
<mergeScheduler class="org.apache.lucene.index.ConcurrentMergeScheduler"/>
```

#### *mergedSegmentWarmer*

When using Solr in for Near Real Time Searching a merged segment warmer can be configured to warm the reader on the newly merged segment, before the merge commits. This is not required for near real-time search, but will reduce search latency on opening a new near real-time reader after a merge completes.

```
<mergedSegmentWarmer class="org.apache.lucene.index.SimpleMergedSegmentWarmer"/>
```

#### *checkIntegrityAtMerge*

If set to `true`, any actions that result in merging segments will first trigger an integrity check using checksums stored in the index segments (if available).  If the checksums are not correct, the merge will fail and throw an Exception.  (defaults to "`false`" for backwards compatibility)

```
<checkIntegrityAtMerge>true</checkIntegrityAtMerge>
```

### Index Locks

#### *lockType*

The LockFactory options specify its implementation.

`lockType=single` uses SingleInstanceLockFactory, and is for a read-only index or when there is no possibility of another process trying to modify the index.

`lockType=native` uses NativeFSLockFactory to specify native OS file locking. Do not use when multiple Solr web applications in the same JVM are attempting to share a single index.

`lockType=simple` uses SimpleFSLockFactory to specify a plain file for locking.

`native` is the default for Solr3.6 and later versions; otherwise `simple` is the default.

For more information on the nuances of each LockFactory, see http://wiki.apache.org/lucene-java/AvailableLockFactories.

```
<lockType>native</lockType>
```

#### *unlockOnStartup*

If **true**, any write or commit locks that have been held will be unlocked on system startup. This defeats the locking mechanism that allows multiple processes to safely access a Lucene index. The default is **false**, and changing this should only be done with care. This parameter is not used if the `lockType` is "none" or "single".

```
<unlockOnStartup>false</unlockOnStartup>
```

#### *`writeLockTimeout`*

The maximum time to wait for a write lock on an IndexWriter. The default is 1000, expressed in milliseconds.

```
<writeLockTimeout>1000</writeLockTimeout>
```

#### Other Indexing Settings

There are a few other parameters that may be important to configure for your implementation. These settings affect how or when updates are made to an index.

| Setting | Description |
|---|---|
| termIndexInterval | Controls how often terms are loaded into memory. The default is 128. |
| reopenReaders | Controls if IndexReaders will be re-opened, instead of closed and then opened, which is often less efficient. The default is true. |
| deletionPolicy | Controls how commits are retained in case of rollback. The default is `SolrDeletionPolicy`, which has sub-parameters for the maximum number of commits to keep (`maxCommitsToKeep`), the maximum number of optimized commits to keep (`maxOptimizedCommitsToKeep`), and the maximum age of any commit to keep (`maxCommitAge`), which supports `DateMathParser` syntax. |
| infoStream | The InfoStream setting instructs the underlying Lucene classes to write detailed debug information from the indexing process as Solr log messages. |

```
<termIndexInterval>128</termIndexInterval>
<reopenReaders>true</reopenReaders>
<deletionPolicy class="solr.SolrDeletionPolicy">
  <str name="maxCommitsToKeep">1</str>
  <str name="maxOptimizedCommitsToKeep">0</str>
  <str name="maxCommitAge">1DAY</str>
</deletionPolicy>
<infoStream>false</infoStream>
```

ⓘ The `maxFieldLength` parameter was removed in Solr 4. If restricting the length of fields is important to you, you can get similar behavior with the `LimitTokenCountFactory`, which can be defined for the fields you'd like to limit. For example, `<filter class="solr.LimitTokenCountFilterFactory" maxTokenCount="10000"/>` would limit the field to 10,000 characters.

## UpdateHandlers in SolrConfig

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.

```
<updateHandler
class="solr.DirectUpdateHandler2">
  ...
</updateHandler>
```

Topics covered in this section:

### Commits

Data sent to Solr is not searchable until it has been *committed* to the index. The reason for this is that in some cases commits can be slow and they should be done in isolation from other possible commit requests to avoid overwriting data. So, it's preferable to provide control over when data is committed. Several options are available to control the timing of commits.

### ***`commit`* and *`softCommit`***

With Solr 4, `commit` is generally used only as a boolean flag sent with a client update request. The command `commit=true` would perform a commit as soon as the data is finished loading to Solr.

You can also set the flag `softCommit=true` to do a 'soft' commit, meaning that Solr will commit your changes quickly but not guarantee that documents are in stable storage. This is an implementation of Near Real Time storage, a feature that boosts document visibility, since you don't have to wait for background merges and storage (to ZooKeeper, if using SolrCloud) to finish before moving on to something else. A full commit means that, if a server crashes, Solr will know exactly where your data was stored; a soft commit means that the data is stored, but the location information isn't yet stored. The tradeoff is that a soft commit gives you faster visibility because it's not waiting for background merges to finish.

For more information about Near Real Time operations, see Near Real Time Searching.

### ***`autoCommit`***

These settings control how often pending updates will be automatically pushed to the index. An alternative to `autoCommit` is to use `commitWithin`, which can be defined when making the update request to Solr (i.e., when pushing documents), or in an update RequestHandler.

| Setting | Description |
|---|---|
| maxDocs | The number of updates that have occurred since the last commit. |
| maxTime | The number of milliseconds since the oldest uncommitted update. |
| openSearcher | Whether to open a new searcher when performing a commit. If this is **false**, the default, the commit will flush recent index changes to stable storage, but does not cause a new searcher to be opened to make those changes visible |

If either of these `maxDocs` or `maxTime` limits are reached, 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.

```
<autoCommit>
  <maxDocs>10000</maxDocs>
  <maxTime>1000</maxTime>
  <openSearcher>false</openSearcher>
</autoCommit>
```

You can also specify 'soft' autoCommits in the same way that you can specify 'soft' commits, except that instead of using `autoCommit` you set the `autoSoftCommit` tag.

```
<autoSoftCommit>
  <maxTime>1000</maxTime>
</autoSoftCommit>
```

#### ***`commitWithin`***

The `commitWithin` settings allow forcing document commits to happen in a defined time period. This is used most frequently with Near Real Time Searching, and for that reason the default is to perform a soft commit. This does not, however, replicate new documents to slave servers in a master/slave environment. If that's a requirement for your implementation, you can force a hard commit by adding a parameter, as in this example:

```
<commitWithin>
  <softCommit>false</softCommit>
</commitWithin>
```

With this configuration, when you call `commitWithin` as part of your update message, it will automatically perform a hard commit every time.

#### ***`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.

```
<maxPendingDeletes>100000</maxPendingDeletes>
```

##### Event Listeners

The UpdateHandler section is also where update-related event listeners can be configured. These can be triggered to occur after a commit or optimize event, or after only an optimize event.

The listener is called with the `RunExecutableListener`, which runs an external executable with the defined set of instructions. The available commands are:

| Setting | Description |
|---|---|

| | |
|---|---|
| event | If **postCommit**, the `RunExecutableListener` will be run after every commit or optimize. If **postOptimize**, the `RunExecutableListener` will be run every optimize only. |
| exe | The name of the executable to run. It should include the path to the file, relative to Solr home. |
| dir | The directory to use as the working directory. The default is ".". |
| wait | Forces the calling thread to wait until the executable returns a response. The default is **true**. |
| args | Any arguments to pass to the program. The default is none. |
| env | Any environment variables to set. The default is none. |

Below is the example from `solrconfig.xml`, which shows an example from script-based replication described at http://wiki.apache.org/solr/CollectionDistribution:

```
<listener event="postCommit" class="solr.RunExecutableListener">
  <str name="exe">solr/bin/snapshooter</str>
  <str name="dir">.</str>
  <bool name="wait">true</bool>
  <arr name="args"> <str>arg1</str> <str>arg2</str> </arr>
  <arr name="env"> <str>MYVAR=val1</str> </arr>
</listener>
```

#### Transaction Log

As described in the section RealTime Get, a transaction log is required for that feature. It is configured in the `updateHandler` section of `solrconfig.xml`.

Realtime Get currently relies on the update log feature, which is enabled by default. It relies on an update log, which is configured in `solrconfig.xml`, in a section like:

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
</updateLog>
```

## Query Settings in SolrConfig

The settings in this section affect the way that Solr will process and respond to queries. These settings are all configured in child elements of the `<query>` element in `solrconfig.xml`.

```
<query>
  ...
</query>
```

Topics covered in this section:

#### Caches

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, there are three cache implementations: `solr.search.LRUCache`, `solr.search.FastLRUCache,` and `solr.search.LFUCache` .

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.

Both `LRUCache` and `FastLRUCache` use an auto-warm count that supports both integers and percentages which get evaluated relative to the current size of the cache when warming happens.

The `LFUCache` refers to the Least Frequently Used cache. This works in a way similar to the LRU cache, except that when the cache fills up, the entry that has been used the least is evicted.

The Statistics page in the Solr Admin UI 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.

Each cache has settings to define it's initial size (`initialSize`), maximum size (`size`) and number of items to use for during warming (`autowarmCount`). The LRU and FastLRU cache implementations can take a percentage instead of an absolute value for `autowarmCount`.

Details of each cache are described below.

### *filterCache*

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 Searching 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 Searching.

```
<filterCache class="solr.LRUCache"
             size="512"
             initialSize="512"
             autowarmCount="128"/>
```

### *queryResultCache*

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.

```
<queryResultCache class="solr.LRUCache"
                  size="512"
                  initialSize="512"
                  autowarmCount="128"/>
```

### *documentCache*

This cache holds Lucene Document objects (the stored fields for each document). Since Lucene internal document IDs are transient, this cache is not auto-warmed. The size for the `documentCache` should always be greater than `max_results` times the `max_concurrent_queries`, to ensure that Solr does not need to refetch a document during a request. The more fields you store in your documents, the higher the memory usage of this cache will be.

```
<documentCache class="solr.LRUCache"
               size="512"
               initialSize="512"
               autowarmCount="0"/>
```

#### *User Defined Caches*

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()`.

```
<cache name="myUserCache" class="solr.LRUCache"
                          size="4096"
                          initialSize="1024"
                          autowarmCount="1024"
                          regenerator="org.mycompany.mypackage.MyRegenerator" />
```

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`. In Solr 4.5, you can also use the `NoOpRegenerator`, which simply repopulates the cache with old items. Define it with the `regenerator` parameter as `"regenerator=solr.NoOpRegenerator"`.

#### Query Sizing and Warming

### *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.

```
<maxBooleanClauses>1024</maxBooleanClauses>
```

> This option modifies a global property that effects all Solr cores. If multiple `solrconfig.xml` files disagree on this property, the value at any point in time will be based on the last Solr core that was initialized.

### *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.

```
<enableLazyFieldLoading>true</enableLazyFieldLoading>
```

### *useFilterForSortedQuery*

This parameter configures Solr to use a filter to satisfy a search. If the requested sort does not include "score", the `filterCache` will be checked for a filter matching the query. For most situations, this is only useful if the same search is requested often with different sort options and none of them ever use "score".

```
<useFilterForSortedQuery>true</useFilterForSortedQuery>
```

### *queryResultWindowSize*

Used with the `queryResultCache`, this will cache a superset of the requested number of document IDs. For example, if the a search in response to a particular query requests documents 10 through 19, and `queryWindowSize` is 50, documents 0 through 49 will be cached.

```
<queryResultWindowSize>20</queryResultWindowSize>
```

### *queryResultMaxDocsCached*

This parameter sets the maximum number of documents to cache for any entry in the `queryResultCache`.

```
<queryResultMaxDocsCached>200</queryResultMaxDocsCached>
```

### *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.

```
<useColdSearcher>false</useColdSearcher>
```

### *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.

```
<maxWarmingSearchers>2</maxWarmingSearchers>
```

#### Query-Related Listeners

As described in the section on #Caches, new Index Searchers are cached. It's possible to use the triggers for listeners to perform query-related tasks. The most common use of this is to define queries to further "warm" the Index Searchers while they are starting. One benefit of this approach is that field caches are pre-populated for faster sorting.

Good query selection is key with this type of listener. It's best to choose your most common and/or heaviest queries and include not just the keywords used, but any other parameters such as sorting or filtering requests.

There are two types of events that can trigger a listener. A `firstSearcher` event occurs when a new searcher is being prepared but there is no current registered searcher to handle requests or to gain auto-warming data from (i.e., on Solr startup). A `newSearcher` event is fired whenever a new searcher is being prepared and there is a current searcher handling requests.

The listener is always instantiated with the class `solr.QuerySenderListener`, and followed a `NamedList` array. These examples are included with `solrconfig.xml`:

```
<listener event="newSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
  <!--
    <lst><str name="q">solr</str><str name="sort">price asc</str></lst>
    <lst><str name="q">rocks</str><str name="sort">weight asc</str></lst>
   -->
  </arr>
</listener>

<listener event="firstSearcher" class="solr.QuerySenderListener">
  <arr name="queries">
    <lst><str name="q">static firstSearcher warming in solrconfig.xml</str></lst>
  </arr>
</listener>
```

> ⚠ The above code sample is the default in `solrconfig.xml`, and a key best practice is to modify these defaults before taking your application to production. While the sample queries are commented out in the section for the "newSearcher", the example is not commented out for the "firstSearcher" event. There is no point in auto-warming your Index Searcher with the query string "static firstSearcher warming in solrconfig.xml" if that is not relevant to your search application.

## RequestDispatcher in SolrConfig

The `requestDispatcher` element of `solrconfig.xml` controls the way the Solr servlet's `RequestDispatcher` implementation responds to HTTP requests. Included are parameters for defining if it should handle `/select` urls (for Solr 1.1 compatibility), if it will support remote streaming, the maximum size of file uploads and how it will respond to HTTP cache headers in requests.

Topics in this section:

### `handleSelect` Element

> ⚠ `handleSelect` is for legacy back-compatibility; those new to Solr do not need to change anything about the way this is configured by default.

The first configurable item is the `handleSelect` attribute on the `<requestDispatcher>` element itself. This attribute can be set to one of two values, either "true" or "false". It governs how Solr responds to requests such as `/select?qt=XXX`. The default value "false" will ignore requests to `{/select` if a requestHandler is not explicitly registered with the name `/select`. A value of "true" will route query requests to the parser defined with the `qt` value.

In recent versions of Solr, a `/select` requestHandler is defined by default, so a value of "false" will work fine. See the section RequestHandlers and SearchComponents in SolrConfig for more information.

```
<requestDispatcher handleSelect="true" >
  ...
</requestDispatcher>
```

### `requestParsers` Element

The `<requestParsers>` sub-element controls values related to parsing requests. This is an empty XML element that doesn't have any content, only attributes.

The attribute `enableRemoteStreaming` controls whether remote streaming of content is allowed. If set to `false`, streaming will not be allowed. Setting it to `true` (the default) 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 in kilobytes 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.

The attribute `formdataUploadLimitInKB` sets a limit in kilobytes on the size of form data (application/x-www-form-urlencoded) submitted in a HTTP POST request, which can be used to pass request parameters that will not fit in a URL.

The attribute `addHttpRequestToContext` can be used to indicate that the original `HttpServletRequest` object should be included in the context map of the `SolrQueryRequest` using the key `httpRequest`. This `HttpServletRequest` is not be used by any Solr components, but may be useful when developing custom plugins.

```
<requestParsers enableRemoteStreaming="true"
                multipartUploadLimitInKB="2048000"
                formdataUploadLimitInKB="2048"
                addHttpRequestToContext="false" />
```

### `httpCaching` Element

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 next 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. |

```
<httpCaching never304="false"
             lastModFrom="openTime"
             etagSeed="Solr">
  <cacheControl>max-age=30, public</cacheControl>
</httpCaching>
```

#### ***`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.

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.

## RequestHandlers and SearchComponents in SolrConfig

After the `<query>` section, request handlers and search components are configured.These are often referred to as "requestHandler" and "searchComponent", which is how they are defined in `solrconfig.xml`.

A *request handler* processes requests coming to Solr. These might be query requests or index update requests. You will likely need several of these defined, depending on how you want Solr to handle the various requests you will make.

A *search component* is a feature of search, such as highlighting or faceting. The search component is defined in `solrconfig.xml` separate from the request handlers, and then registered with a request handler as needed.

Topics covered in this section:

### Request Handlers

Every request handler is defined with a name and a class. The name of the request handler is referenced with the request to Solr. For example, a request to `http://localhost:8983/solr/collection1` is the default address for Solr, which will likely bring up the Solr Admin UI. However, add "/select" to the end, you can make a query:

```
http://localhost:8983/solr/collection1/select?q=solr
```

This query will be processed by the request handler with the name "/select". We've only used the "q" parameter here, which includes our query term, a simple keyword of "solr". If the request handler has more parameters defined, those will be used with any query we send to this request handler unless they are over-ridden by the client (or user) in the query itself.

If you have another request handler defined, you would send your request with that name - for example, "/update" is a request handler that handles index updates like sending new documents to the index.

#### *SearchHandlers*

The primary request handler defined with Solr by default is the "SearchHandler", which handles search queries. The request handler is defined, and then a list of defaults for the handler are defined with a `defaults` list.

For example, in the default `solrconfig.xml`, the first request handler defined looks like this:

```
<requestHandler name="/select" class="solr.SearchHandler">
     <lst name="defaults">
       <str name="echoParams">explicit</str>
       <int name="rows">10</int>
       <str name="df">text</str>
     </lst>
</requestHandler>
```

This example defines the `rows` parameter, which defines how many search results to return, to "10". The default field to search is the "text" field, set with the `df` parameter. The `echoParams` parameter defines that the parameters defined in the query should be returned when debug information is returned. Note also that the way the defaults are defined in the list varies if the parameter is a string, an integer, or another type.

All of the parameters described in the section on searching can be defined as defaults for any of the SearchHandlers.

Besides `defaults`, there are other options for the SearchHandler, which are:

- `appends`: This allows definition of parameters that are added to the user query. These might be filter queries, or other query rules that should be added to each query. There is no mechanism in Solr to allow a client to override these additions, so you should be absolutely sure you always want these parameters applied to queries.

  ```
  <lst name="appends">
     <str name="fq">inStock:true</str>
  </lst>
  ```

  In this example, the filter query "inStock:true" will always be added to every query.
- `invariants`: This allows definition of parameters that cannot be overridden by a client. The values defined in an `invariants` section will always be used regardless of the values specified by the user, by the client, in `defaults` or in `appends`.

  ```
  <lst name="invariants">
     <str name="facet.field">cat</str>
     <str name="facet.field">manu_exact</str>
     <str name="facet.query">price:[* TO 500]</str>
     <str name="facet.query">price:[500 TO *]</str>
  </lst>
  ```

  In this example, facet fields have been defined which limits the facets that will be returned by Solr. If the client requests facets, the facets defined with a configuration like this are the only facets they will see.

The final section of a request handler definition is `components`, which defines a list of search components that can be used with a request handler. They are only registered with the request handler. How to define a search component is discussed further on in the section on Search Components. The `components` element can only be used with a request handler that is a SearchHandler.

The `solrconfig.xml` file includes many other examples of SearchHandlers that can be used or modified as needed.

#### *UpdateRequestHandlers*

The UpdateRequestHandlers are request handlers which process updates to the index.

In this guide, we've covered these handlers in detail in the section Uploading Data with Index Handlers.

#### *ShardHandlers*

It is possible to configure a request handler to search across shards of a cluster, used with distributed search. More information about distributed search and how to configure the shardHandler is in the section Distributed Search with Index Sharding.

#### *Other Request Handlers*

There are other request handlers defined in `solrconfig.xml`, covered in other sections of this guide:

- RealTime Get
- Index Replication
- Ping

#### Search Components

The search components define the logic that is used by the SearchHandler to perform queries for users.

##### *Default Components*

There are several defaults search components that work with all SearchHandlers without any additional configuration. If no components are defined, these are used by default.

| Component Name | Class Name | More Information |
|---|---|---|
| query | solr.QueryComponent | Described in the section Query Syntax and Parsing. |
| facet | solr.FacetComponent | Described in the section Faceting. |
| mlt | solr.MoreLikeThisComponent | Described in the section MoreLikeThis. |
| highlight | solr.HighlightComponent | Described in the section Highlighting. |
| stats | solr.StatsComponent | Described in the section The Stats Component. |
| debug | solr.DebugComponent | Described in the section on Common Query Parameters. |

If you register a new search component with one of these default names, the newly defined component will be used instead of the default.

##### *First-Components and Last-Components*

It's possible to define some components as being used before (with `first-components`) or after (with `last-components`) other named components. This would be useful if custom search components have been configured to process data before the regular components are used. This is used when registering the components with the request handler.

```
<arr name="first-components">
   <str>mycomponent</str>
</arr>

<arr name="components">
   <str>query</str>
   <str>facet</str>
   <str>mlt</str>
   <str>highlight</str>
   <str>spellcheck</str>
   <str>stats</str>
   <str>debug</str>
</arr>
```

##### *Other Useful Components*

Many of the other useful components are described in sections of this Guide for the features they support. These are:

- `SpellCheckComponent`, described in the section Spell Checking.
- `TermVectorComponent`, described in the section The Term Vector Component.
- `QueryElevationComponent`, described in the section The Query Elevation Component.
- `TermsComponent`, described in the section The Terms Component.

#### Related Topics

- SolrRequestHandler from the Solr Wiki.
- SearchHandler from the Solr Wiki.
- SearchComponent from the Solr Wiki.

## Solr Cores and solr.xml

`solr.xml` has evolved from configuring one Solr core to supporting multiple Solr cores and eventually to defining parameters for SolrCloud. Particularly with the advent of SolrCloud, the ability to cleanly define and maintain high-level configuration parameters in solr.xml Solr cores has become more difficult so an alternative is being adopted.

Starting in Solr 4.3, Solr will maintain two distinct formats for `solr.xml`, the *legacy* and *discovery* modes. The former is the format we have become accustomed to in which all of the cores one wishes to define in a Solr instance are defined in `solr.xml` in `<cores><core/>...<core/></cores>` tags. This format will continue to be supported through the entire 4.x code line.

As of Solr 5.0 this form of `solr.xml` will no longer be supported. Instead Solr will support *core discovery*. In brief, core discovery still defines some configuration parameters in `solr.xml`, but *no cores are defined in this file*. Instead, the solr home directory is recursively walked until a `core.properties` file is encountered. This file is presumed to be at the root of a core, and many of the options that were placed in the `<core>` tag in legacy Solr are now defined here as simple properties, i.e. a file with entries, one to a line, like '`name=core1`', '`schema=myschema.xml`' and so on.

In Solr 4.x, the presence of a `<solr><cores>` node determines whether Solr uses legacy or discovery mode. There are checks at initialization time. If one tries to mix legacy and discovery tags in solr.xml. Solr will refuse to initialize if "mixed mode" is discovered, and errors will be logged.

> ⚠ The new "core discovery mode" structure for `solr.xml` will become mandatory as of Solr 5.0, see: Format of solr.xml.

The following links are to pages that define these options in more detail, giving the acceptable parameters for the legacy and discovery modes.

- **Format of solr.xml**: The new *discovery* mode for `solr.xml`, including the acceptable parameters in both the `solr.xml` file and the corresponding `core.properties` files.
- **Legacy solr.xml Configuration**: The *legacy* mode for `solr.xml` and the acceptable parameters.
- **Moving to the New solr.xml Format**: How to migrate from legacy to discovery `solr.xml` configurations.
- **CoreAdmin API**: Tools and commands for core administration, which is common to both legacy and discovery modes.

### Format of solr.xml

You can find `solr.xml` in your Solr Home directory. The default *discovery* `solr.xml` file looks like this:

```
<solr>

  <solrcloud>
    <str name="host">${host:}</str>
    <int name="hostPort">${jetty.port:8983}</int>
    <str name="hostContext">${hostContext:solr}</str>
    <int name="zkClientTimeout">${zkClientTimeout:15000}</int>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
  </solrcloud>

  <shardHandlerFactory name="shardHandlerFactory"
    class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>

</solr>
```

As you can see, the discovery solr configuration is "SolrCloud friendly". However, the presence of the `<solrcloud>` element does *not* mean that the Solr instance is running in SolrCloud mode. Unless the `-DzkHost` or `-DzkRun` are specified at startup time, this section is ignored.

#### Using Multiple SolrCores

It is possible to segment Solr into multiple 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 Solr cores on the fly, shutdown cores, even replace one running core with another, all without ever stopping or restarting your servlet container.

Solr cores are configured by placing a file named `core.properties` in a sub-directory under `solr.home`. There are no a-priori limits to the depth of the tree, nor are there limits to the number of cores that can be defined. Cores may be anywhere in the tree with the exception that cores may *not* be defined under an existing core. That is, the following is not allowed:

```
./cores/core1/core.properties
./cores/core1/coremore/core5/core.properties
```

In this example, the enumeration will stop at "core1".

The following is legal:

```
./cores/somecores/core1/core.properties
./cores/somecores/core2/core.properties
./cores/othercores/core3/core.properties
./cores/extracores/deepertree/core4/core.properties
```

A minimal `core.properties` file looks like this:

```
name=collection1
```

This is very different than the legacy `solr.xml` `<core>` tag. In fact, your `core.properties` file *can be empty*. Say the `core.properties` file is located in (relative to `solr_home`) `./cores/core1`. In that case, the file core name is assumed to be "core1". The instanceDir will be the folder containing `core.properties` (i.e., `./cores/core1`). The dataDir will be `../cores/core1/data`, etc.

ⓘ You can run Solr without configuring any cores.

#### Solr.xml Parameters

##### *The `<solr>` Element*

There are no attributes that you can specify in the `<solr>` tag, which is the root element of `solr.xml`. The tables below list the child nodes of each XML element in `solr.xml`.

⚠ The `persistent` attribute is no longer supported in `solr.xml`. The properties in `solr.xml` are immutable, and any changes to individual cores are persisted in the individual `core.properties` files.

| Node | 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 more information on this parameter, see the Solr Wiki at http://wiki.apache.org/solr/CoreAdmin#cores. |
| `collectionsHandler` | As above, for custom CollectionsHandler implementations |
| `infoHandler` | As above, for custom InfoHandler implementations |
| `coreLoadThreads` | Specifies the number of threads that will be assigned to load cores in parallel |
| `coreRootDirectory` | The root of the core discovery tree, defaults to SOLR_HOME |
| `managementPath` | no-op at present. |
| `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. |
| `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. |
| `transientCacheSize` | Defines how many cores with transient=true that can be loaded before swapping the least recently used core for a new core. |
| `configSetBaseDir` | The directory under which configsets for solr cores can be found. Defaults to SOLR_HOME/configsets |

##### *The `<solrcloud>` element*

This element defines several parameters that relate so SolrCloud. This section is ignored unless the solr instance is started with either `-DzkRun` or `-DzkHost`

| Node | Description |
|---|---|
| `distribUpdateConnTimeout` | Used to set the underlying "connTimeout" for intra-cluster updates. |
| `distribUpdateSoTimeout` | Used to set the underlying "socketTimeout" for intra-cluster updates. |
| `host` | The hostname Solr uses to access cores. |
| `hostContext` | The servlet context path. |
| `hostPort` | The port Solr uses to access cores. In the default `solr.xml` file, this is set to `${jetty.port:}`, which will use the Solr port defined in Jetty. |
| `leaderVoteWait` | When SolrCloud is starting up, how long each Solr node will wait for all known replicas for that shard to be found before assuming that any nodes that haven't reported are down. |
| `leaderConflictResolveWait` | When trying to elect a leader for a shard, this property sets the maximum time a replica will wait to see conflicting state information to be resolved; temporary conflicts in state information can occur when doing rolling restarts, especially when the node hosting the Overseer is restarted. Typically, the default value of 180000 (millis) is sufficient for conflicts to be resolved; you may need to increase this value if you have hundreds or thousands of small collections in SolrCloud. |
| `zkClientTimeout` | A timeout for connection to a ZooKeeper server. It is used with SolrCloud. |
| `zkHost` | In SolrCloud mode, the URL of the ZooKeeper host that Solr should use for cluster state information. |
| `genericCoreNodeNames` | If `TRUE`, node names are not based on the address of the node, but on a generic name that identifies the core. When a different machine takes over serving that core things will be much easier to understand. |

##### *The `<logging>` element*

| Node | Description |
|---|---|
| `class` | The class to use for logging. The corresponding JAR file must be available to solr, perhaps through a `<lib>` directive in solrconfig.xml. |
| `enabled` | true/false - whether to enable logging or not. |

##### The `<logging><watcher>` element

| Node | Description |
|---|---|
| `size` | The number of log events that are buffered. |
| `threshold` | The logging level above which your particular logging implementation will record. For example when using log4j one might specify DEBUG, WARN, INFO, etc. |

#### *The `<shardHandlerFactory>` element*

Custom shard handlers can be defined in `solr.xml` if you wish to create a custom shard handler.

```
<shardHandlerFactory name="ShardHandlerFactory" class="qualified.class.name">
```

However, since this is a custom shard handler, sub-elements are specific to the implementation.

#### Substituting JVM System Properties in `solr.xml`

Solr supports variable substitution of JVM system property values in `solr.xml`, which allows runtime specification of various configuration options. The syntax is `${propertyname[:option default value]}`. This allows defining a default that can be overridden when Solr is launched. If a default value is not specified, then the property must be specified at runtime or the solr.xml file will generate an error when parsed.

Any JVM System properties, usually specified using the -D flag when starting the JVM, can be used as variables in the `solr.xml` file.

For example: In the `solr.xml` file shown below, starting solr using `java -DsocketTimeout=1000 -jar start.jar` will cause the `socketTimeout` option of the `HttpShardHandlerFactory` to be overridden using a value of 1000ms, instead of the default property value of "0" – however the `connTimeout` option will continue to use the default property value of "0".

```
<solr>
  <shardHandlerFactory name="shardHandlerFactory"
                       class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:0}</int>
    <int name="connTimeout">${connTimeout:0}</int>
  </shardHandlerFactory>
</solr>
```

#### Individual `core.properties` Files

Core discovery replaces the individual `<core>` tags in `solr.xml` with a `core.properties` file located on disk. The presence of the `core.properties` file *defines* the `instanceDir` for that core. The `core.properties` file is a simple Java Properties file where each line is just a key=value pair, e.g., `name=core1`. Notice that no quotes are required.

The minimal `core.properties` file is an empty file, in which case all of the properties are defaulted appropriately.

Java properties files allow the hash ("#") or bang ("!") characters to specify comment-to-end-of-line. This table defines the recognized properties:

| key | Description |
|---|---|
| `name` | The name of the SolrCore. You'll use this name to reference the SolrCore when running commands with the CoreAdminHandler. |
| `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` | Core's data directory as a path relative to the instanceDir, `data` by default. |

| | |
|---|---|
| `configSet` | If set, the name of the configset to use to configure the core (see Config Sets). |
| `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`. |
| `transient` | If **true**, the core can be unloaded if Solr reaches the `transientCacheSize`. The default if not specified is **false**. Cores are unloaded in order of least recently used first. *Setting to **true** is not recommended in SolrCloud mode.* |
| `loadOnStartup` | If **true**, the default if it is not specified, the core will loaded when Solr starts. *Setting to **false** is not recommended in SolrCloud mode.* |
| `coreNodeName` | Added in Solr 4.2, this attributes allows naming a core. The name can then be used later if you need to replace a machine with a new one. By assigning the new machine the same coreNodeName as the old core, it will take over for the old SolrCore. |
| `ulogDir` | The absolute or relative directory for the update log for this core (SolrCloud) |
| `shard` | The shard to assign this core to (SolrCloud) |
| `collection` | The name of the collection this core is part of (SolrCloud) |
| `roles` | Future param for SolrCloud or a way for users to mark nodes for their own use. |

Additional "user defined" properties may be specified for use as variables in parsing core configuration files.

## Legacy solr.xml Configuration

Use `solr.xml` to configure your Solr core (a logical index and associated configuration files), or to configure multiple cores. You can find `solr.xml` in your Solr Home directory. The default `solr.xml` file looks like this:

```
<solr persistent="true">
  <cores adminPath="/admin/cores" defaultCoreName="collection1" host="${host:}"
         hostPort="${jetty.port:}" hostContext="${hostContext:}"
         zkClientTimeout="${zkClientTimeout:15000}">
    <core name="collection1" instanceDir="collection1" />
  </cores>
</solr>
```

For more information about core configuration and `solr.xml`, see http://wiki.apache.org/solr/CoreAdmin.

### Using Multiple SolrCores

It is possible to segment Solr into multiple 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 Solr cores on the fly, shutdown cores, even replace one running core with another, all without ever stopping or restarting your servlet container.

Solr cores are configured by placing a file named `solr.xml` in your `solr.home` directory. A typical `solr.xml` looks like this:

```
<solr persistent="false">
  <cores adminPath="/admin/cores" host="${host:}" hostPort="${jetty.port:}">
    <core name="core0" instanceDir="core0" />
    <core name="core1" instanceDir="core1" />
  </cores>
</solr>
```

This sets up two Solr cores, named "core0" and "core1", and names the directories (relative to the Solr installation path) which will store the configuration and data sub-directories.

> You can run Solr without configuring any cores.

### Solr.xml Parameters

#### *The `<solr>` Element*

There are several attributes that you can specify on `<solr>`, which is the root element of `solr.xml`.

| Attribute | Description |
|---|---|
| `coreLoadThreads` | Specifies the number of threads that will be assigned to load cores in parallel |
| `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. The default is true. |
| `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. |
| `zkHost` | In SolrCloud mode, the URL of the ZooKeeper host that Solr should use for cluster state information. |

> 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.

#### *The `<cores>` Element*

The `<cores>` element, which contains definitions for each Solr core, is a child of `<solr>` and accepts several 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. |
| `host` | The hostname Solr uses to access cores. |
| `hostPort` | The port Solr uses to access cores. In the default `solr.xml` file, this is set to `${jetty.port:}`, which will use the Solr port defined in Jetty. |
| `hostContext` | The servlet context path. |
| `zkClientTimeout` | A timeout for connection to a ZooKeeper server. It is used with SolrCloud. |
| `distribUpdateConnTimeout` | Used to set the underlying "connTimeout" for intra-cluster updates. |
| `distribUpdateSoTimeout` | Used to set the underlying "socketTimeout" for intra-cluster updates |
| `leaderVoteWait` | When SolrCloud is starting up, how long each Solr node will wait for all known replicas for that share to be found before assuming that any nodes that haven't reported are down. |
| `genericCoreNodeNames` | If `TRUE`, node names are not based on the address of the node, but on a generic name that identifies the core. When a different machine takes over serving that core things will be much easier to understand. |
| `managementPath` | no-op at present. |
| `defaultCoreName` | The name of a core that will be used for requests that do not specify a core. |
| `transientCacheSize` | Defines how many cores with `transient=true` that can be loaded before swapping the least recently used core for a new core. |
| `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`. |
| `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 more information on this parameter, see the Solr Wiki at http://wiki.apache.org/solr/CoreAdmin#cores. |

#### ***The `<logging>` Element***

There is at most one `<logging>` element for a Solr installation that defines various attributes for logging.

| Attribute | Description |
|---|---|

|  |  |
|---|---|
| `class` | The class to use for logging. The corresponding JAR file must be available to solr, perhaps through a <lib> directive in solrconfig.xml. |
| `enabled` | true/false - whether to enable logging or not. |

In addition, the `<logging>` element may have a child element `<watcher>` which may have the following attributes

|  |  |
|---|---|
| `size` | The number of log events that are buffered. |
| `threshold` | The logging level above which your particular logging implementation will record. For example when using log4j one might specify DEBUG or WARN or INFO etc. |

#### *The <core> Element*

There is one `<core>` element for each SolrCore you define. They are children of the `<cores>` element and each one accepts the following 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 the 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 the 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`. |
| `transient` | If **true**, the core can be unloaded if Solr reaches the `transientCacheSize`. The default if not specified is **false**. Cores are unloaded in order of least recently used first. |
| `loadOnStartup` | If **true**, the default if it is not specified, the core will loaded when Solr starts. |
| `coreNodeName` | Added in Solr 4.2, this attributes allows naming a core. The name can then be used later if you need to replace a machine with a new one. By assigning the new machine the same coreNodeName as the old core, it will take over for the old SolrCore. |
| `ulogDir` | The absolute or relative directory for the update log for this core (SolrCloud) |
| `shard` | The shard to assign this core to (SolrCloud) |
| `collection` | The name of the collection this core is part of (SolrCloud) |
| `roles` | Future param for SolrCloud or a way for users to mark nodes for their own use. |

#### *Substituting JVM System Properties in `solr.xml`*

Solr supports variable substitution of JVM system property values in `solr.xml`, which allows runtime specification of various configuration options. The syntax is `${propertyname[:option default value]}`. This allows defining a default that can be overridden when Solr is launched. If a default value is not specified, then the property must be specified at runtime or the solr.xml file will generate an error when parsed.

Any JVM System properties, usually specified using the -D flag when starting the JVM, can be used as variables in the `solr.xml` file.

For example: In the `solr.xml` file shown below, starting solr using `java -Dmy.logging=true -jar start.jar` will cause the `enabled` option of the log watcher to be overridden using a value of `true`, instead of the default property value of "false" – however the `threshold` option will continue to use the default property value of "INFO".

```
<solr persistent="true">
  <logging enabled="${my.logging:false}">
    <watcher size="100" threshold="${my.logging.level:INFO}" />
  </logging>
  <cores adminPath="/admin/cores">
    <core name="collection1" instanceDir="collection1" />
  </cores>
</solr>
```

### *User Defined Properties in `solr.xml`*

You can define custom 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.

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.

```
<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>
```

## Moving to the New solr.xml Format

Migration from old-style `solr.xml` to core discovery is very straightforward. First, modify the `solr.xml` file from the legacy format to the discovery format.

In general there is a direct analog from the legacy format to the new format *except* there is no `<cores>` element nor are there any `<core>` elements in discovery-based Solr.

### Startup

In Solr 4.4 and on, the presence of a `<cores>` child element of the `<solr>` element in the `solr.xml` file signals a legacy version of `solr.xml`, and cores are expected to be defined as they have been historically. Depending on whether a `<cores>` element is discovered, `solr.xml` is parsed as either a legacy or discovery file and errors are thrown in the log if legacy and discovery modes are mixed in `solr.xml`.

### Moving `<core>` definitions.

To migrate to discovery-based `solr.xml`, remove all of the `<core>` elements and the enclosing `<cores>` element from `solr.xml`. See the pages linked above for examples of migrating other attributes. Then, in the instanceDir for each core create a `core.properties` file. *This file can be empty if all defaults are acceptable*. In particular, the `instanceDir` is assumed to be the directory in which the `core.properties` file is discovered. The data directory will be in a directory called "data" directly below. If the file is completely empty, the name of the core is assumed to be the name of the folder in which the `core.properties` file was discovered.

As mentioned elsewhere, the tree structure that the cores are in is arbitrary, with the exception that the directories containing the `core.properties` files must share a common root, but that root may be many levels up the tree. Note that supporting a root for the cores that is not a child of `SOLR_HOME` is supported through properties in `solr.xml`. However, only *one* root is possible, there is no provision presently for specifying multiple roots.

The only restriction on the tree structure is that cores may not be children of other cores; enumeration stops descending *down* the tree when the first `core.properties` file is discovered. Siblings of the directory in which the `core.properties` file is discovered are still walked, only stopping recursing down the sibling when a `core.properties` file is found.

### Example

Here's an example of what a legacy `solr.xml` file might look like and the equivalent discovery-based `solr.xml` and `core.properties` files:

```
<solr persistent="${solr.xml.persist:false}">
  <cores adminPath="/admin/cores" defaultCoreName="collection1" host="127.0.0.1"
hostPort="${hostPort:8983}"
         hostContext="${hostContext:solr}"
zkClientTimeout="${solr.zkclienttimeout:30000}" shareSchema="${shareSchema:false}"
         genericCoreNodeNames="${genericCoreNodeNames:true}">
    <core name="core1" instanceDir="core1" shard="${shard:}"
collection="${collection:core1}" config="${solrconfig:solrconfig.xml}"
schema="${schema:schema.xml}" coreNodeName="${coreNodeName:}"/>
    <core name="core2" instanceDir="core2" />
    <shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
      <int name="socketTimeout">${socketTimeout:120000}</int>
      <int name="connTimeout">${connTimeout:15000}</int>
    </shardHandlerFactory>
  </cores>
</solr>
```

The new-style `solr.xml` might look like what is below. Note that adminPath, defaultCoreName are not supported in discovery-based solr.xml.

```
<solr>
  <solrcloud>
    <str name="host">127.0.0.1</str>
    <int name="hostPort">${hostPort:8983}</int>
    <str name="hostContext">${hostContext:solr}</str>
    <int name="zkClientTimeout">${solr.zkclienttimeout:30000}</int>
    <str name="shareSchema">${shareSchema:false}</str>
    <bool name="genericCoreNodeNames">${genericCoreNodeNames:true}</bool>
  </solrcloud>

  <shardHandlerFactory name="shardHandlerFactory" class="HttpShardHandlerFactory">
    <int name="socketTimeout">${socketTimeout:120000}</int>
    <int name="connTimeout">${connTimeout:15000}</int>
  </shardHandlerFactory>
</solr>
```

In each of "core1" and "core2" directories, there would be a `core.properties` file that might look like these. Note that note that instanceDir is not supported, it is assumed to be the directory in which core.properties is found.

core1:

```
name=core1
shard=${shard:}
collection=${collection:core1}
config=${solrconfig:solrconfig.xml}
schema=${schema:schema.xml}
coreNodeName=${coreNodeName:}
```

core2:

```
name=core2
```

In fact, the core2 `core.properties` file could even be empty and the name would default to the directory in which the `core.properties` file was found.

## CoreAdmin API

The CoreAdminHandler is a special SolrRequestHandler that is used to manage Solr cores. 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 Solr core administration.

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, and are defined in depth in the sections below.

- STATUS
- CREATE
- RELOAD
- RENAME
- SWAP

- UNLOAD
- MERGEINDEXES
- SPLIT
- REQUESTSTATUS

### STATUS

The STATUS action returns the status of all running Solr cores, or status for only the named core.

```
http://localhost:8983/solr/admin/cores?action=STATUS&core=core0
```

***Input***

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | No | | The name of a core, as listed in the "name" attribute of a `<core>` element in `solr.xml`. |
| indexInfo | boolean | No | true | If **false**, information about the index will not be returned with a core STATUS request. In Solr implementations with a large number of cores (i.e., more than hundreds), retrieving the index information for each core can take a lot of time and isn't always required. |

### 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 Solr core 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/di
r&config=config_file_name.xml&schema=schem_file_name.xml&dataDir=data
```

***Input***

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| name | string | Yes | N/A | The name of the new core. Same as "name" on the `<core>` element. |
| instanceDir | string | Yes | N/A | The directory where files for this SolrCore should be stored. Same as `instanceDir` on the `<core>` element. |
| config | string | No | | Name of the config file (solrconfig.xml) relative to `instanceDir`. |
| schema | string | No | | Name of the schema file (schema.xml) relative to `instanceDir`. |
| datadir | string | No | | Name of the data directory relative to `instanceDir`. |
| configSet | string | No | | Name of the configset to use for this core (see Config Sets) |

| | | | | |
|---|---|---|---|---|
| collection | string | No | | The name of the collection to which this core belongs. The default is the name of the core. `collection.<param>=<value>` causes a property of `<param>=<value>` to be set if a new collection is being created. Use `collection.configName=<configname>` to point to the configuration for a new collection. |
| shard | string | No | | The shard id this core represents. Normally you want to be auto-assigned a shard id. |
| property.*name*=*value* | string | No | | Sets the core property *name* to *value*. See core.properties file contents. |
| async | string | No | | Request ID to track this action which will be processed asynchronously |

Use `collection.configName=<configname>` to point to the config for a new collection.

#### *Example*

```
http://localhost:8983/solr/admin/cores?action=CREATE&name=mycore&collection=collection1&shard=shard2
```

### RELOAD

The `RELOAD` action loads a new core from the configuration of an existing, registered Solr core. While the new core is initializing, the existing one will continue to handle requests. When the new Solr core is ready, it takes over and the old core is unloaded.

This is useful when you've made changes to a Solr core'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. However the Core Container does not persist the SolrCloud `solr.xml` parameters, such as `solr/@zkHost` and `solr/cores/@hostPort`, which are ignored.

```
http://localhost:8983/solr/admin/cores?action=RELOAD&core=core0
```

As of Solr 4.0, RELOAD performs "live" reloads of SolrCore, reusing some existing objects. Some configuration options, such as the `DataDir` location and `IndexWriter` related settings in `solrconfig.xml` can not be changed and made active with a simple RELOAD action.

#### *Input*

#### Query Parameters

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes | N/A | The name of the core, as listed in the "name" attribute of a `<core>` element in `solr.xml`. |

### RENAME

The `RENAME` action changes the name of a Solr core.

```
http://localhost:8983/solr/admin/cores?action=RENAME&core=core0&other=core5
```

#### *Input*

### Query Parameters

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes | | The name of the Solr core to be renamed. |
| other | string | Yes | | The new name for the Solr core. 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. |
| async | string | No | | Request ID to track this action which will be processed asynchronously |

### `SWAP`

`SWAP` atomically swaps the names used to access two existing Solr cores. 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
```

> ⚠ Do not use `SWAP` with a SolrCloud node. It is not supported and can result in the core being unusable.

## Input

### Query Parameters

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes | | The name of one of the cores to be swapped. |
| other | string | Yes | | The name of one of the cores to be swapped. |
| async | string | No | | Request ID to track this action which will be processed asynchronously |

### `UNLOAD`

The `UNLOAD` action removes a core from 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 (`core`) identifying the core 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`.

> ⚠ Unloading all cores in a SolrCloud collection causes the removal of that collection's metadata from ZooKeeper.

#### *Input*

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes |  | The name of one of the cores to be removed. |
| deleteIndex | boolean | No | false | If true, will remove the index when unloading the core. |
| deleteDataDir | boolean | No | false | If true, removes the `data` directory and all sub-directories. |
| deleteInstanceDir | boolean | No | false | If true, removes everything related to the core, including the index directory, configuration files and other related files. |
| async | string | No |  | Request ID to track this action which will be processed asynchronously |

### `MERGEINDEXES`

The `MERGEINDEXES` action merges one or more indexes to another index. The indexes must have completed commits, and should be locked against writes until the merge is complete or the resulting merged index may become corrupted. The target core index must already exist and have a compatible schema with the one or more indexes that will be merged to it. Another commit on the target core should also be performed after the merge is complete.

```
http://localhost:8983/solr/admin/cores?action=MERGEINDEXES&core=core0&indexDir=/o
pt/solr/core1/data/index&indexDir=/opt/solr/core2/data/index
```

In this example, we use the `indexDir` parameter to define the index locations of the source cores. The `core` parameter defines the target index. A benefit of this approach is that we can merge any Lucene-based index that may not be associated with a Solr core.

Alternatively, we can instead use a `srcCore` parameter, as in this example:

```
http://localhost:8983/solr/admin/cores?action=mergeindexes&core=core0&srcCore=cor
e1&srcCore=core2
```

This approach allows us to define cores that may not have an index path that is on the same physical server as the target core. However, we can only use Solr cores as the source indexes. Another benefit of this approach is that we don't have as high a risk for corruption if writes occur in parallel with the source index.

We can make this call run asynchronously by specifying the `async` parameter and passing a request-id. This id can then be used to check the status of the already submitted task using the REQUESTSTATUS API.

#### *Input*

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes |  | The name of the target core/index. |
| indexDir | string |  |  | Multi-valued, directories that would be merged. |
| srcCore | string |  |  | Multi-valued, source cores that would be merged. |

| | | | | |
|---|---|---|---|---|
| async | string | | | Request ID to track this action which will be processed asynchronously |

### SPLIT

The SPLIT action splits an index into two or more indexes. The index being split can continue to handle requests. The split pieces can be placed into a specified directory on the server's filesystem or it can be merged into running Solr cores.

The SPLIT action supports five parameters, which are described in the table below.

***Input***

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| core | string | Yes | | The name of the core to be split. |
| path | string | | | Multi-valued, the directory path in which a piece of the index will be written. |
| targetCore | string | | | Multi-valued, the target Solr core to which a piece of the index will be merged |
| ranges | string | No | | A comma-separated list of hash ranges in hexadecimal format |
| split.key | string | No | | The key to be used for splitting the index |
| async | string | No | | Request ID to track this action which will be processed asynchronously |

> ⚠ Either `path` or `targetCore` parameter must be specified but not both. The ranges and split.key parameters are optional and only one of the two should be specified, if at all required.

***Examples***

The `core` index will be split into as many pieces as the number of `path` or `targetCore` parameters.

**Usage with two `targetCore` parameters:**

```
http://localhost:8983/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1&targetCore=core2
```

Here the `core` index will be split into two pieces and merged into the two `targetCore` indexes.

**Usage of with two `path` parameters:**

```
http://localhost:8983/solr/admin/cores?action=SPLIT&core=core0&path=/path/to/index/1&path=/path/to/index/2
```

The `core` index will be split into two pieces and written into the two directory paths specified.

**Usage with the `split.key` parameter:**

```
http://localhost:8983/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1&s
plit.key=A!
```

Here all documents having the same route key as the `split.key` i.e. 'A!' will be split from the `core` index and written to the `targetCore`.

**Usage with ranges parameter:**

```
http://localhost:8983/solr/admin/cores?action=SPLIT&core=core0&targetCore=core1&t
argetCore=core2&targetCore=core3&ranges=0-1f4,1f5-3e8,3e9-5dc
```

This example uses the `ranges` parameter with hash ranges 0-500, 501-1000 and 1001-1500 specified in hexadecimal. Here the index will be split into three pieces with each targetCore receiving documents matching the hash ranges specified i.e. core1 will get documents with hash range 0-500, core2 will receive documents with hash range 501-1000 and finally, core3 will receive documents with hash range 1001-1500. At least one hash range must be specified. Please note that using a single hash range equal to a route key's hash range is NOT equivalent to using the `split.key` parameter because multiple route keys can hash to the same range.

The `targetCore` must already exist and must have a compatible schema with the `core` index. A commit is automatically called on the `core` index before it is split.

This command is used as part of the SPLITSHARD command but it can be used for non-cloud Solr cores as well. When used against a non-cloud core without `split.key` parameter, this action will split the source index and distribute its documents alternately so that each split piece contains an equal number of documents. If the `split.key` parameter is specified then only documents having the same route key will be split from the source index.

### REQUESTSTATUS

Request the status of an already submitted asynchronous CoreAdmin API call.

***Input***

**Query Parameters**

| Parameter | Type | Required | Default | Description |
|---|---|---|---|---|
| requestid | string | Yes | | The user defined request-id for the Asynchronous request. |

The call below will return the status of an already submitted Asynchronous CoreAdmin call.

```
http://localhost:8983/solr/admin/cores?action=REQUESTSTATUS&requestid=1
```

## Config Sets

On a multicore Solr instance, you may find that you want to share configuration between a number of different cores.  You can achieve this using named configsets, which are essentially shared configuration directories stored under a configurable configset base directory.

To create a configset, simply add a new directory under the configset base directory.  The configset will be identified by the name of this directory.  Then into this copy the config directory you want to share.  The structure should look something like this:

```
/<configSetBaseDir>
 /configset1
  /conf
   /schema.xml
   /solrconfig.xml
 /configset2
  /conf
   /schema.xml
   /solrconfig.xml
```

The default base directory is `SOLR_HOME/configsets`, and it can be configured in solr.xml.

To create a new core using a configset, pass `configSet` as one of the core properties.  For example, via the core admin API:

```
http://<solr>/cores?action=CREATE&name=mycore&instanceDir=path/to/instance&configSet=configset2
```

## 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.

Not everyone will need to create plugins for their Solr instances - what's provided is usually enough for most applications. However, if there's something that you need, you may want to review the Solr Wiki documentation on plugins at SolrPlugins.

## 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.

For more general information about improving Solr performance, see https://wiki.apache.org/solr/SolrPerformanceFactors.

### 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 robust hardware, you may get better results running multiple JVMs, rather than just one with a large memory heap. Some specialized JVM implementations may have customized garbage collection algorithms that do better with large heaps. Also, Java 7 is expected 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 indexes, 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: Oracle's Java HotSpot Garbage Collection.

### Use the Server HotSpot VM

If you are using Sun's JVM, add the `-server` command-line option when you start 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 is 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.

### 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 Managing Solr for more information.

# Managing Solr

This section describes how to run Solr and how to look at Solr when it is running. It contains the following sections:

Running Solr on Jetty: Describes how to run Solr in the Jetty web application container. The Solr example included in this distribution runs in a Jetty web application container.

Running Solr on Tomcat: Describes how to run Solr in the Tomcat web application container.

Configuring Logging: Describes how to configure logging for Solr.

Enabling SSL: Describes how to configure single-node Solr and SolrCloud to encrypt internal and external communication using SSL.

Backing Up: Describes backup strategies for your Solr indexes.

Using JMX with Solr: Describes how to use Java Management Extensions with Solr.

Managed Resources: Describes the REST APIs for dealing with resources that various Solr plugins may expose.

Running Solr on HDFS: How to use HDFS to store your Solr indexes and transaction logs.

For information on running Solr in a variety of Java application containers, see the basic installation instructions on the Solr wiki.

# Running Solr on Tomcat

Solr comes with an example schema and scripts for running on Jetty. The next section describes some of the details of how things work "under the hood," and covers running multiple Solr instances and deploying Solr using the Tomcat application manager.

For more information about running Solr on Tomcat, see the basic installation instructions and the Solr Tomcat page on the Solr wiki.

## How Solr Works with Tomcat

The two basic steps for running Solr in any Web application container are as follows:

1. 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 Solr, take a look in `tomcat/webapps`:you'll see the `solr.war` file is already there.
2. 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"
   ```

Port 8983 is the default Solr listening port. If you are using Tomcat and wish to change this port, edit the file `tomcat/conf/server.xml` in the Solr 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.

> ⚠ 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.

### Running Multiple Solr Instances

The standard way to deploy multiple Solr index instances in a single Web application is to use the multicore API described in Solr Cores and solr.xml.

An alternative approach, which provides more code isolation, uses Tomcat context fragments. A context fragment is a file that contains a single `<context>` element and any subelements required for your application. The file omits all other XML elements.

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`.

In Tomcat's `conf/Catalina/localhost` directory, store one context fragment per instance of Solr. If the `conf/Catalina/localhost` directory doesn't exist, go ahead and create it.

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` directory:

**harvey.xml (http://localhost:8983/harvey using /some/path/solr1home)**

```
<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>
```

**rupert.xml (http://localhost:8983/rupert using /some/path/solr2home)**

```
<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>
```

### Deploying Solr with the Tomcat Manager

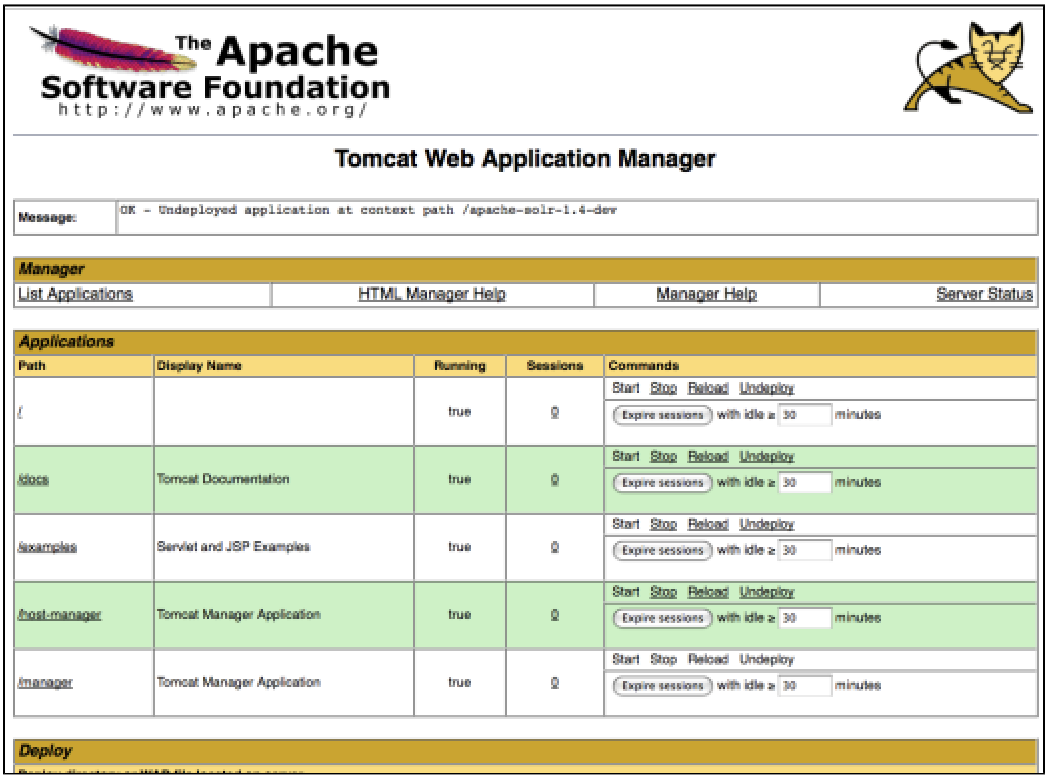
If your instance of Tomcat is running the Tomcat Web Application Manager, you can use its browser interface to deploy Solr.

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 will see the main screen of the manager.



The Apache Software Foundation
http://www.apache.org/

Tomcat Web Application Manager

| | |
|---|---|
| Message: | `OK - Undeployed application at context path /apache-solr-1.4-dev` |

**Manager**

| | | | |
|---|---|---|---|
| List Applications | HTML Manager Help | Manager Help | Server Status |

**Applications**

| Path | Display Name | Running | Sessions | Commands |
|---|---|---|---|---|
| / | | true | 0 | Start Stop Reload Undeploy<br>Expire sessions with idle ≥ 30 minutes |
| /docs | Tomcat Documentation | true | 0 | Start Stop Reload Undeploy<br>Expire sessions with idle ≥ 30 minutes |
| /examples | Servlet and JSP Examples | true | 0 | Start Stop Reload Undeploy<br>Expire sessions with idle ≥ 30 minutes |
| /host-manager | Tomcat Manager Application | true | 0 | Start Stop Reload Undeploy<br>Expire sessions with idle ≥ 30 minutes |
| /manager | Tomcat Manager Application | true | 0 | Start Stop Reload Undeploy<br>Expire sessions with idle ≥ 30 minutes |

**Deploy**

To add Solr, scroll down to the **Deploy** section, specifically **WAR file to deploy**. Click **Browse...** and find the Solr WAR file, usually something like `dist/solr-4.x.y.war` within your Solr installation. Click **Deploy**. Tomcat will load the WAR file and start running it. Click 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.

## Running Solr on Jetty

Solr comes with an example schema and scripts for running on Jetty, along with a working installation, in the `/example` directory. It is stripped of all unnecessary features and its config has had some minor tuning so it's optimized for Solr. It is recommended that you use the provided Jetty server for optimal performance. For more information about the Jetty example installation, see the Solr Tutorial and the basic installation instructions.

For detailed information about running Solr on a stand-alone Jetty, see http://wiki.apache.org/solr/SolrJetty.

### Change the Solr Listening Port

Port 8983 is the default port for Solr. If you are using Jetty and wish to change the port number, edit the file `example/etc/jetty.xml` in the Solr distribution. You'll find the port in this part of the file:

```
<New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
  <Set name="host"><SystemProperty name="jetty.host" /></Set>
  <Set name="port"><SystemProperty name="jetty.port" default="8983"/></Set>
  <Set name="maxIdleTime">50000</Set>
  <Set name="Acceptors">2</Set>
  <Set name="statsOn">false</Set>
  <Set name="confidentialPort">8443</Set>
  <Set name="lowResourcesConnections">5000</Set>
  <Set name="lowResourcesMaxIdleTime">5000</Set>
</New>
```

Modify the port number as desired and restart Jetty if it is already running.

> Modifying the port number will leave some of the samples and help file links pointing to the wrong 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.

## Configuring Logging

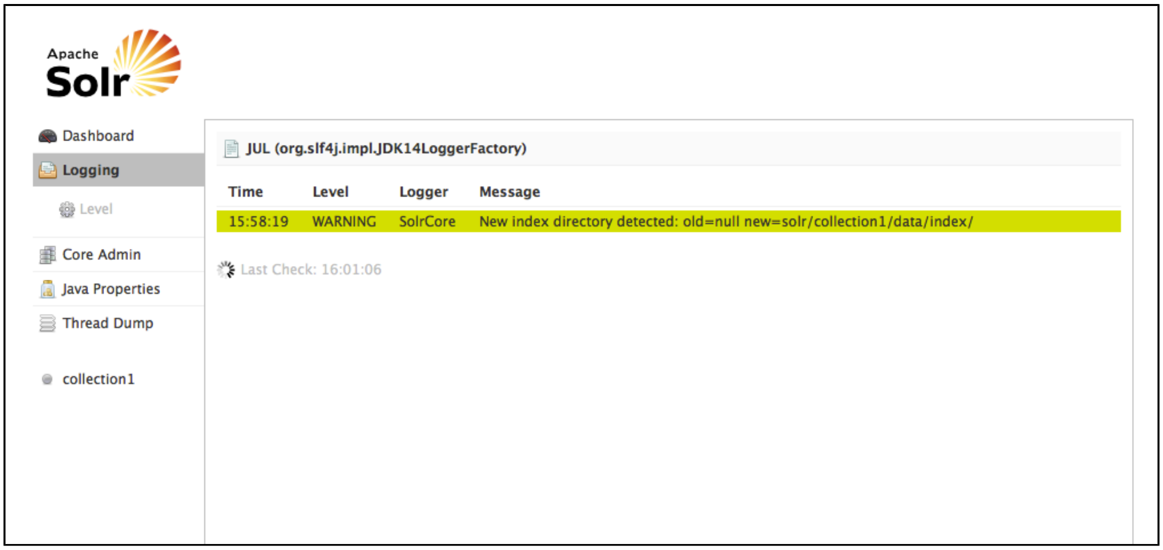
Prior to version 4.3, Solr used the SLF4J Logging API (http://www.slf4j.org). To improve flexibility in logging with containers other than Jetty, in Solr 4.3 the default behavior has changed and the SLF4J jars were removed from Solr's `.war` file. This allows changing or upgrading the logging mechanism as needed.

For further information about Solr logging, see SolrLogging.

> In addition to the logging options described below, there is a way to configure which request parameters (such as parameters sent as part of queries) are logged with an additional request parameter called `logParamsList`. See the section on Common Query Parameters for more information.

### 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 Using the Solr Administration User Interface.)



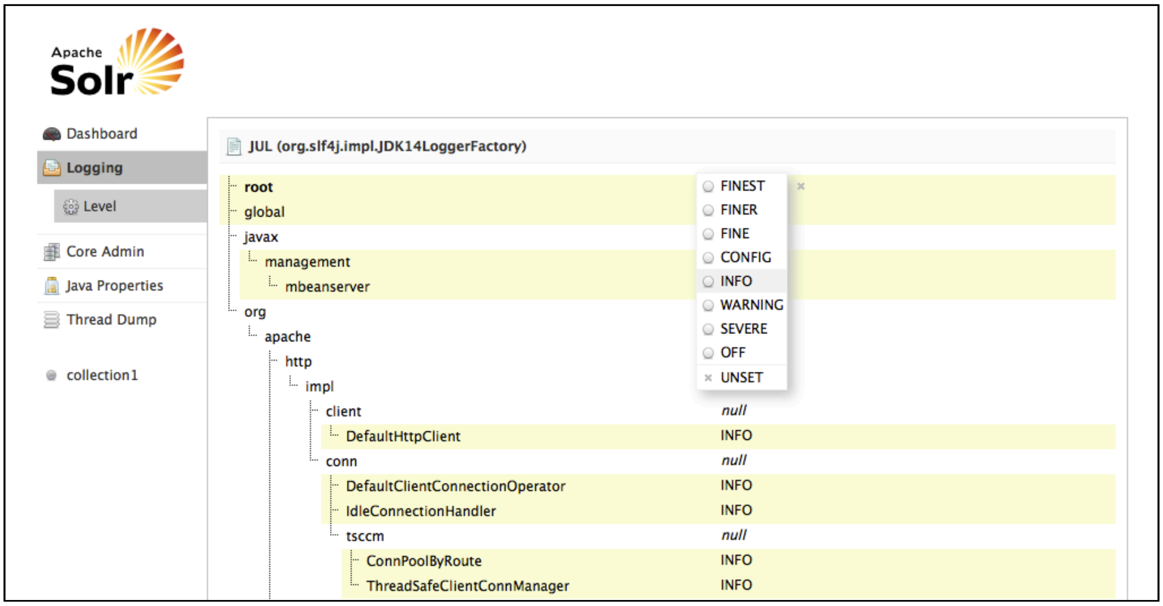
JUL (org.slf4j.impl.JDK14LoggerFactory)

| Time | Level | Logger | Message |
|---|---|---|---|
| 15:58:19 | WARNING | SolrCore | New index directory detected: old=null new=solr/collection1/data/index/ |

*The Logging 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.

When you select **Level**, you see the following menu:



JUL (org.slf4j.impl.JDK14LoggerFactory)

```
root
global
javax
  management
    mbeanserver
org
  apache
    http
      impl
        client                                   null
          DefaultHttpClient                      INFO
        conn                                     null
          DefaultClientConnectionOperator        INFO
          IdleConnectionHandler                  INFO
          tsccm                                  null
            ConnPoolByRoute                      INFO
            ThreadSafeClientConnManager          INFO
```

*The Log Level Menu.*

Directories are shown with their current logging levels. The Log Level Menu floats over these. To set a log level for a particular directory, select it and click the appropriate log level button.

Log levels settings are as follows:

| Level | Result |
|---|---|
| FINEST | Reports everything. |
| FINE | Reports everything but the least important messages. |
| CONFIG | Reports configuration errors. |
| INFO | Reports everything but normal status. |
| WARNING | Reports all warnings. |
| SEVERE | Reports only the most severe warnings. |
| OFF | Turns off logging. |
| UNSET | Removes the previous log setting. |

Multiple settings at one time are allowed.

### Permanent Logging Settings

Making permanent changes to the JDK Logging API configuration is a matter of creating or editing a properties file.

#### Tomcat Logging Settings

Tomcat offers a choice between settings for all applications or settings specifically for the Solr application.

With Solr 4.3, you will need to copy the SLF4J `.jar` files from the `example/lib/ext` directory to the main `lib` directory of Tomcat (this may be as simple as `tomcat/lib`). Then you can copy the `log4j.properties` file from `example/resources` to a location on the classpath - the same location as the `.jar` files is probably OK in most cases. Then you can edit the properties as needed to set the log destination.

See the documentation for the SLF4J Logging API for more information:

http://slf4j.org/docs.html

http://docs.oracle.com/javase/7/docs/technotes/guides/logging/index.html

#### 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` or modifying the one found in `example/etc`.

To tell Jetty how to find the file, edit `jetty.xml` and add the following property information:

```
<Configure id="Server" class="org.mortbay.jetty.Server">
   <Call class="java.lang.System" name="setProperty">
      <Arg>java.util.logging.config.file</Arg>
      <Arg>logging.properties</Arg>
   </Call>
</Configure>
```

The next time you launch Jetty, it will use the settings in the file.

## Enabling SSL

Both SolrCloud and single-node Solr can encrypt communications to and from clients, and in SolrCloud between nodes, with SSL.  This section describes enabling SSL with the example Jetty server using a self-signed certificate.

For background on SSL certificates and keys, see http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/.

### Basic SSL Setup

#### Generate a self-signed certificate and a key

To generate a self-signed certificate and a single key that will be used to authenticate both the server and the client, we'll use the JDK `keytool` command and create a separate keystore. This keystore will also be used as a truststore below. It's possible to use the keystore that comes with the JDK for these purposes, and to use a separate truststore, but those options aren't covered here.

Run the commands below in the `example/etc/` directory in the binary Solr distribution.

The "`-ext SAN=...`" `keytool` option allows you to specify all the DNS names and/or IP addresses that will be allowed during hostname verification (but see below for how to skip hostname verification between Solr nodes so that you don't have to specify all hosts here). In addition to `localhost` and `127.0.0.1`, this example includes a LAN IP address `192.168.1.3` for the machine the Solr nodes will be running on:

```
keytool -genkeypair -alias solr-ssl -keyalg RSA -keysize 2048 -keypass secret
-storepass secret -validity 9999 -keystore solr-ssl.keystore.jks -ext
SAN=DNS:localhost,IP:192.168.1.3,IP:127.0.0.1 -dname "CN=localhost, OU=Organizational
Unit, O=Organization, L=Location, ST=State, C=Country"
```

The above command will create a keystore file named `solr-ssl.keystore.jks` in the current directory.

#### Convert the certificate and key to PEM format for use with cURL

cURL isn't capable of using JKS formatted keystores, so the JKS keystore needs to be converted to PEM format, which cURL understands.

First convert the JKS keystore into PKCS12 format using `keytool`:

```
keytool -importkeystore -srckeystore solr-ssl.keystore.jks -destkeystore
solr-ssl.keystore.p12 -srcstoretype jks -deststoretype pkcs12
```

Next convert the PKCS12 format keystore, including both the certificate and the key, into PEM format using the `openssl` command:

```
openssl pkcs12 -in solr-ssl.keystore.p12 -out solr-ssl.pem
```

#### Configure Jetty

The example directory in the Solr binary distribution contains a Jetty server configured to run Solr in non-SSL mode out of the box. The configuration changes below will allow Jetty to communicate using SSL with the keystore prepared above.

First, comment out the non-SSL `SelectChannelConnector` block in `example/etc/jetty.xml` using **`<!--`** before and **`-->`** afterward:

```
<!--
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <Set name="host"><SystemProperty name="jetty.host" /></Set>
      <Set name="port"><SystemProperty name="jetty.port" default="8983"/></Set>
      <Set name="maxIdleTime">50000</Set>
      <Set name="Acceptors">2</Set>
      <Set name="statsOn">false</Set>
      <Set name="confidentialPort">8443</Set>
      <Set name="lowResourcesConnections">5000</Set>
      <Set name="lowResourcesMaxIdleTime">5000</Set>
    </New>
  </Arg>
</Call>
-->
```

Next, uncomment the `SslSelectChannelConnector` block by removing the **`<!--`** before and **`-->`** afterward, and change the keyStore value to point to the JKS keystore created above - the result should look like this:

```
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
      <Arg>
        <New class="org.eclipse.jetty.http.ssl.SslContextFactory">
          <Set name="keyStore"><SystemProperty name="jetty.home"
default="."/>/etc/solr-ssl.keystore.jks</Set>
          <Set name="keyStorePassword">secret</Set>
          <Set name="needClientAuth"><SystemProperty name="jetty.ssl.clientAuth"
default="false"/></Set>
        </New>
      </Arg>
      <Set name="port"><SystemProperty name="jetty.ssl.port" default="8984"/></Set>
      <Set name="maxIdleTime">30000</Set>
    </New>
  </Arg>
</Call>
```

#### Run Single Node Solr using SSL

The command below, run from the `example/` directory in the binary Solr distribution, will start Solr on port 8984.  By default clients will not be required to authenticate:

```
java -jar start.jar
```

Alternatively, to require clients to authenticate, you can set the `jetty.ssl.clientAuth` system property to `true` (default is `false`):

```
java -Djetty.ssl.clientAuth=true -jar start.jar
```

### SolrCloud

This section describes how to run a two-node SolrCloud cluster with no initial collections and a single-node external

ZooKeeper.  The commands below assume you have already created the keystore described above.

### Configure ZooKeeper

> ZooKeeper does not support encrypted communication with clients like Solr.  There are several related JIRA tickets where SSL support is being planned/worked on: ZOOKEEPER-235; ZOOKEEPER-236; ZOOKEEPER-733; and  ZOOKEEPER-1000.

Before you start any SolrCloud nodes, you must configure your solr cluster properties in ZooKeeper, so that Solr nodes know to communicate via SSL.

This section assumes you have created and started a single-node external ZooKeeper on port 2181 on localhost - see Setting Up an External ZooKeeper Ensemble

The `urlScheme` cluster-wide property needs to be set to `https` before any Solr node starts up.  The example below uses the `zkcli.sh` client that comes with the binary Solr distribution to do this, from the `example/` directory:

```
scripts/cloud-scripts/zkcli.sh -zkhost localhost:2181 -cmd put /clusterprops.json
'{"urlScheme":"https"}'
```

### Run SolrCloud with SSL

#### ***Copy the `example/` directory***

Create two copies of the `example/` directory and remove the `collection1/` directories - from the root directory of the binary Solr distribution:

```
cp -r example node1
rm -rf node1/solr/collection1
cp -r example node2
rm -rf node2/solr/collection1
```

#### ***Start the first Solr node***

Next, start the first Solr node on port 8984 and bootstrap a configset we'll call "myconfig" (taken from the `example/solr/collection1/conf/` directory):

```
cd node1
java -DzkHost=localhost:2181 -Djetty.port=8984 -Djetty.ssl.port=8984
-Dbootstrap_confdir=../example/solr/collection1/conf -Dcollection.configName=myconf
-Djavax.net.ssl.keyStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.keyStorePassword=secret
-Djavax.net.ssl.trustStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStorePassword=secret -jar start.jar
```

Alternatively, if you created your SSL key without all DNS names/IP addresses on which Solr nodes will run, you can tell Solr to skip hostname verification for inter-Solr-node communications by setting the `solr.ssl.checkPeerName` system property to `false`:

```
cd node1
java -Dsolr.ssl.checkPeerName=false -DzkHost=localhost:2181 -Djetty.port=8984
-Djetty.ssl.port=8984 -Dbootstrap_confdir=../example/solr/collection1/conf
-Dcollection.configName=myconf -Djavax.net.ssl.keyStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.keyStorePassword=secret
-Djavax.net.ssl.trustStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStorePassword=secret -jar start.jar
```

#### *Start the second Solr node*

Finally, start the second Solr node on port 7574 - again, to skip hostname verification, add `-Dsolr.ssl.checkPeerName=false` (not shown here):

```
cd node2
java -DzkHost=localhost:2181 -Djetty.port=7574 -Djetty.ssl.port=7574
-Djavax.net.ssl.keyStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.keyStorePassword="secret"
-Djavax.net.ssl.trustStore=etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStorePassword="secret" -jar start.jar
```

Note that both the `jetty.port` and `jetty.ssl.port` system properties are required when starting SolrCloud using SSL.

### Example Client Actions

> ⚠ cURL on OS X Mavericks has degraded SSL support. For more information and workarounds to allow 1-way SSL, see http://curl.haxx.se/mail/archive-2013-10/0036.html

#### Create a SolrCloud collection using cURL

Create a 2-shard, rf=1 collection named mycollection, from a directory containing the PEM formatted certificate and key created above (e.g. `example/etc/`) - this command will perform client authentication using the same key as the Solr nodes; if you have not enabled client authentication (system property `-Djetty.ssl.clientAuth=true`), then you can remove the `-E solr-ssl.pem:secret` option:

```
curl -E solr-ssl.pem:secret --cacert solr-ssl.pem
"https://localhost:8984/solr/admin/collections?action=CREATE&name=mycollection&numShar
ds=2&replicationFactor=1&maxShardsPerNode=1&collection.configName=myconf"
```

This should return an XML-formatted response showing successful collection creation.

#### Retrieve SolrCloud cluster status using cURL

To get the resulting cluster status (again, if you have not enabled client authentication, remove the `-E solr-ssl.pem:secret` option):

```
curl -E solr-ssl.pem:secret --cacert solr-ssl.pem
"https://localhost:8984/solr/admin/collections?action=CLUSTERSTATUS&wt=json&indent=on"
```

You should get a response that looks like this:

```
{
  "responseHeader":{
    "status":0,
    "QTime":2041},
  "cluster":{
    "collections":{
      "mycollection":{
        "shards":{
          "shard1":{
            "range":"80000000-ffffffff",
            "state":"active",
            "replicas":{"core_node1":{
                "state":"active",
                "base_url":"https://127.0.0.1:8984/solr",
                "core":"mycollection_shard1_replica1",
                "node_name":"127.0.0.1:8984_solr",
                "leader":"true"}}},
          "shard2":{
            "range":"0-7fffffff",
            "state":"active",
            "replicas":{"core_node2":{
                "state":"active",
                "base_url":"https://127.0.0.1:7574/solr",
                "core":"mycollection_shard2_replica1",
                "node_name":"127.0.0.1:7574_solr",
                "leader":"true"}}}},
        "maxShardsPerNode":"1",
        "router":{"name":"compositeId"},
        "replicationFactor":"1"}},
    "properties":{"urlScheme":"https"}}}
```

#### Index documents using `post.jar`

Use `post.jar` to index some example documents to the SolrCloud collection created above:

```
cd example/exampledocs
java -Djavax.net.ssl.keyStorePassword=secret
-Djavax.net.ssl.keyStore=../etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStore=../etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStorePassword=secret
-Durl=https://localhost:8984/solr/mycollection/update -jar post.jar *.xml
```

#### Query using cURL

Use cURL to query the SolrCloud collection created above, from a directory containing the PEM formatted certificate and key created above (e.g. `example/etc/`) - if you have not enabled client authentication (system property `-Djetty.ssl.clientAuth=true)`, then you can remove the `-E solr-ssl.pem:secret` option:

```
curl -E solr-ssl.pem:secret --cacert solr-ssl.pem
"https://localhost:8984/solr/mycollection/select?q=*:*&wt=json&indent=on"
```

#### Index a document using `CloudSolrServer`

From a java client using Solrj, index a document.  In the code below, the `javax.net.ssl.*` system properties are set programmatically, but you could instead specify them on the java command line, as in the `post.jar` example above:

```
System.setProperty("javax.net.ssl.keyStore", "/path/to/solr-ssl.keystore.jks");
System.setProperty("javax.net.ssl.keyStorePassword", "secret");
System.setProperty("javax.net.ssl.trustStore", "/path/to/solr-ssl.keystore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "secret");
String zkHost = "127.0.0.1:2181";
CloudSolrServer server = new CloudSolrServer(zkHost);
server.setDefaultCollection("mycollection");
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "1234");
doc.addField("name", "A lovely summer holiday");
server.add(doc);
server.commit();
```

## Backing Up

If you are 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.

### Making Backups with the Solr Replication Handler

The easiest way to make back-ups in Solr is to take advantage of the Replication Handler, which is described in detail in Index Replication. 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 have 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 Legacy Scaling and Distribution.

## 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://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html.

### Configuring JMX

JMX configuration is provided in `solrconfig.xml`. Please see the JMX Technology Home Page for more details.

A `rootName` attribute can be used when configuring `<jmx />` in `solrconfig.xml`. If this attribute is set, Solr uses it as the root name for all the MBeans that Solr exposes via JMX. The default name is "solr" followed by the core name.

> ⚠ Enabling/disabling JMX and securing access to MBeanServers is left up to the user by specifying appropriate JVM parameters and configuration. Please explore the JMX Technology Home Page for more details.

#### Configuring an Existing MBeanServer

The command:

```
<jmx />
```

enables JMX support in Solr if and only if an existing MBeanServer is found. Use this if you want to configure JMX with JVM parameters. Remove this to disable exposing Solr configuration and statistics to JMX. If this is specified, Solr will try to list all available MBeanServers and use the first one to register MBeans.

#### Configuring an Existing MBeanServer with agentId

The command:

```
<jmx agentId="myMBeanServer" />
```

enables JMX support in Solr if and only if an existing MBeanServer is found matching the given agentId. If multiple servers are found, the first one is used. If none is found, an exception is raised and depending on the configuration, Solr may refuse to start.

#### Configuring a New MBeanServer

The command:

```
<jmx serviceUrl="service:jmx:rmi:///jndi/rmi://localhost:9999/solrjmx" />
```

creates a new MBeanServer exposed for remote monitoring at the specific service URL. If the JMXConnectorServer can't be started (probably because the serviceUrl is bad), an exception is thrown.

#### *Example*

Using the example jetty setup provided with Solr installation, the JMX support works like this in `jconsole.png`.

1. Run "ant example" to build the example war file.
2. Go to the example folder in the Solr installation and run the following command:

   ```
   java -Dcom.sun.management.jmxremote -jar start.jar
   ```

3. Start `jconsole` (provided with the Sun JDK in the bin directory).
4. Connect to the "start.jar" shown in the list of local processes.
5. Switch to the "MBeans" tab. You should be able to see "solr" listed there.

#### Configuring a Remote Connection to Solr JMX

If you want to connect to Solr remotely, you need to pass in some extra parameters, documented here:

http://docs.oracle.com/javase/7/docs/technotes/guides/management/agent.html

If you are not able to connect from a remote machine, you may also need to specify the hostname of the Solr host by adding the following property as well:

> ⚠ Making JMX connections into machines running behind NATs (e.g. Amazon's EC2 service) is not a simple task. The `java.rmi.server.hostname` system property may help, but running `jconsole` on the server itself and using a remote desktop is often the simplest solution. See http://web.archive.org/web/20130525022506/http://jmsbrdy.com/monitoring-java-applications-running-on-ec2-i.

## Managed Resources

Managed resources expose a REST API endpoint for performing Create-Read-Update-Delete (CRUD) operations on a Solr object. Any long-lived Solr object that has configuration settings and/or data is a good candidate to be a managed resource.  Managed resources complement other programmatically manageable components in Solr, such as the RESTful schema API to add fields to a managed schema. Consider a Web-based UI that offers Solr-as-a-Service where users need to configure a set of stop words and synonym mappings as part of an initial setup process for their search application. This type of use case can easily be supported using the Managed Stop Filter & Managed Synonym Filter Factories provided by Solr, via the Managed resources REST API.  Users can also write their own custom plugins, that leverage the same internal hooks to make additional resources REST managed.

### Overview

Let's begin learning about managed resources by looking at a couple of examples provided by Solr for managing stop words and synonyms using a REST API. After reading this section, you'll be ready to dig into the details of how managed resources are implemented in Solr so you can start building your own implementation.

#### Stop words

To begin, you need to define a field type that uses the ManagedStopFilterFactory , such as:

```
<fieldType name="managed_en" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ManagedStopFilterFactory"
            managed="english" />
  </analyzer>
</fieldType>
```

There are two important things to notice about this field type definition. First, the filter implementation class is **`solr.ManagedStopFilterFactory`** . This is a special implementation of the StopFilterFactory that uses a set of stop words that are managed from a REST API. Second, the **`managed=”english”`** attribute gives a name to the set of managed stop words, in this case indicating the stop words are for English text.

The REST endpoint for managing the English stop words in the example collection is: `/solr/collection1/schema/analysis/stopwords/english`

The example resource path should be mostly self-explanatory. It should be noted that the ManagedStopFilterFactory implementation determines the **/schema/analysis/stopwords** part of the path, which makes sense because this is an analysis component defined by the schema. It follows that a field type that uses the following filter:

```
<filter class="solr.ManagedStopFilterFactory"
        managed="french" />
```

would resolve to path:  `/solr/collection1/schema/analysis/stopwords/french`

So now let’s see this API in action, starting with a simple GET request:

```
curl "http://localhost:8983/solr/collection1/schema/analysis/stopwords/english"
```

Assuming you sent this request to the example server, the response body is a JSON document:

```
{
  "responseHeader":{
    "status":0,
    "QTime":1
  },
  "wordSet":{
    "initArgs":{"ignoreCase":true},
    "initializedOn":"2014-03-28T20:53:53.058Z",
    "managedList":[
      "a",
      "an",
      "and",
      "are",
      ... ]
  }
}
```

The collection1 core in the example server ships with a built-in set of managed stop words, see: `example/solr/collection1/conf/_schema_analysis_stopwords_english.json`. However, you should only interact with this file using the API and not edit it directly.

One thing that should stand out to you in this response is that it contains a **`managedList`** of words as well as **`initArgs`** . This is an important concept in this framework—managed resources typically have configuration and data. For stop words, the only configuration parameter is a boolean that determines whether to ignore the case of tokens during stop word filtering (ignoreCase=true|false). The data is a list of words, which is represented as a JSON array named **`managedList`** in the response.

Now, let’s add a new word to the English stop word list using an HTTP PUT:

```
curl -X PUT -H 'Content-type:application/json' --data-binary '["foo"]'
"http://localhost:8983/solr/collection1/schema/analysis/stopwords/english"
```

Here we’re using cURL to PUT a JSON list containing a single word “foo” to the managed English stop words set. Solr will return 200 if the request was successful. You can also put multiple words in a single PUT request.

You can test to see if a specific word exists by sending a GET request for that word as a child resource of the set, such as:

```
curl "http://localhost:8983/solr/collection1/schema/analysis/stopwords/english/foo"
```

This request will return a status code of 200 if the child resource (foo) exists or 404 if it does not exist the managed list.

To delete a stop word, you would do:

```
curl -X DELETE
"http://localhost:8983/solr/collection1/schema/analysis/stopwords/english/foo"
```

Note: PUT/POST is used to add terms to an existing list instead of replacing the list entirely. This is because it is more common to add a term to an existing list than it is to replace a list altogether, so the API favors the more common approach of incrementally adding terms especially since deleting individual terms is also supported.

### Synonyms

For the most part, the API for managing synonyms behaves similar to the API for stop words, except instead of working with a list of words, it uses a map, where the value for each entry in the map is a set of synonyms for a term. As with stop words, the example server ships with a minimal set of English synonym mappings that is activated by the following field type definition in schema.xml:

```
<fieldType name="managed_en" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.ManagedStopFilterFactory"
            managed="english" />

    <filter class="solr.ManagedSynonymFilterFactory"
            managed="english" />

  </analyzer>
</fieldType>
```

To get the map of managed English synonyms, send a GET request to:

```
curl "http://localhost:8983/solr/collection1/schema/analysis/synonyms/english"
```

This request will return a response that looks like:

```
{
  "responseHeader":{
    "status":0,
    "QTime":4},
  "synonymMappings":{
    "initArgs":{
      "ignoreCase":true,
      "format":"solr"},
    "initializedOn":"2014-03-31T15:46:48.77Z",
    "managedMap":{
      "gb":["gib","gigabyte"],
      "happy":["glad","joyful"],
      "tv":["television"]
    }
  }
}
```

Managed synonyms are returned under the **managedMap** property which contains a JSON Map where the value of each entry is a set of synonyms for a term, such as happy has synonyms glad and joyful in the example above.

To add a new synonym mapping, you can PUT/POST a single mapping such as:

```
curl -X PUT -H 'Content-type:application/json' --data-binary
'{"mad":["angry","upset"]}'
"http://localhost:8983/solr/collection1/schema/analysis/synonyms/english"
```

The API will return status code 200 if the PUT request was successful. To determine the synonyms for a specific term, you send a GET request for the child resource, such as `/schema/analysis/synonyms/english/mad` would return `["angry","upset"]`. Lastly, you can delete a mapping by sending a DELETE request to the managed endpoint.

### Applying Changes

Changes made to managed resources via this REST API are not applied to the active Solr components until the Solr collection (or Solr core in single server mode) is reloaded. For example:, after adding or deleting a stop word, you must reload the core/collection before changes become active.

This approach is required when running in distributed mode so that we are assured changes are applied to all cores in a collection at the same time so that behavior is consistent and predictable. It goes without saying that you don’t want one of your replicas working with a different set of stop words or synonyms  than the others.

One subtle outcome of this *apply-changes-at-reload* approach is that the once you make changes with the API, there is no way to read the active data. In other words, the API returns the most up-to-date data from an API perspective, which could be different than what is currently being used by Solr components. However, the intent of this API implementation is that changes will be applied using a reload within a short time frame after making them so the time in which the data returned by the API differs from what is active in the server is intended to be negligible.

> ⚠ Changing things like stop words and synonym mappings typically require re-indexing existing documents if being used by index-time analyzers. The RestManager framework does not guard you from this, it simply makes it possible to programmatically build up a set of stop words, synonyms etc.

### RestManager Endpoint

Metadata about registered ManagedResources is available using the **/schema/managed** and **/config/managed** endpoints. Assuming you have the **managed_en** field type shown above defined in your schema.xml, sending a GET request to the following resource will return metadata about which schema-related resources are being managed by the RestManager:

```
curl "http://localhost:8983/solr/collection1/schema/managed"
```

The response body is a JSON document containing metadata about managed resources under the /schema root:

```
{
  "responseHeader":{
    "status":0,
    "QTime":3
  },
  "managedResources":[
    {
      "resourceId":"/schema/analysis/stopwords/english",
      "class":"org.apache.solr.rest.schema.analysis.ManagedWordSetResource",
      "numObservers":"1"
    },
    {
      "resourceId":"/schema/analysis/synonyms/english",

"class":"org.apache.solr.rest.schema.analysis.ManagedSynonymFilterFactory$SynonymManag
er",
      "numObservers":"1"
    }
  ]
}
```

You can also create new managed resource using PUT/POST to the appropriate URL – before ever configuring anything that uses these resources.

For example: imagine we want to build up a set of German stop words. Before we can start adding stop words, we need to create the endpoint:

`/solr/collection1/schema/analysis/stopwords/german`

To create this endpoint, send the following PUT/POST request to the endpoint we wish to create:

```
curl -X PUT -H 'Content-type:application/json' --data-binary \
'{"class":"org.apache.solr.rest.schema.analysis.ManagedWordSetResource"}' \
"http://localhost:8983/solr/collection1/schema/analysis/stopwords/german"
```

Solr will respond with status code 200 if the request is successful. Effectively, this action registers a new endpoint for a managed resource in the RestManager. From here you can start adding German stop words as we saw above:

```
curl -X PUT -H 'Content-type:application/json' --data-binary '["die"]' \
"http://localhost:8983/solr/collection1/schema/analysis/stopwords/german"
```

For most users, creating resources in this way should never be necessary, since managed resources are created automatically when configured.

However: You may want to explicitly delete managed resources if they are no longer being used by a Solr component.

For instance, the managed resource for German that we created above can be deleted because there are no Solr components that are using it, whereas the managed resource for English stop words cannot be deleted because there is a token filter declared in schema.xml that is using it.

```
curl -X DELETE
"http://localhost:8983/solr/collection1/schema/analysis/stopwords/german"
```

### Related Topics

- Using Solr's REST APIs to manage stop words and synonyms by Tim Potter @ SearchHub.org

## Running Solr on HDFS

Solr has support for writing and reading its index and transaction log files to the HDFS distributed filesystem. This does not use Hadoop Map-Reduce to process Solr data, rather it only uses the HDFS filesystem for index and transaction log file storage.

### Basic Configuration

To use HDFS rather than a local filesystem, you must be using Hadoop 2.0.x and configure `solrconfig.xml` properly.

- You need to use an HdfsDirectoryFactory and a data dir of the form `hdfs://host:port/path`
- You need to specify an UpdateLog location of the form `hdfs://host:port/path`
- You should specify a lock factory type of '`hdfs`' or none.

With the default configuration files, you can start Solr on HDFS with the following command:

```
java -Dsolr.directoryFactory=HdfsDirectoryFactory
     -Dsolr.lock.type=hdfs
     -Dsolr.data.dir=hdfs://host:port/path
     -Dsolr.updatelog=hdfs://host:port/path -jar start.jar
```

### SolrCloud Configuration

In SolrCloud mode, it's best to leave the data and update log directories as the defaults Solr comes with and simply specify the solr.hdfs.home. All dynamically created collections will create the appropriate directories automatically under the solr.hdfs.home root directory.

- Set solr.hdfs.home in the form `hdfs://host:port/path`
- You should specify a lock factory type of '`hdfs`' or none.

With the default configuration files, you can start SolrCloud on HDFS with the following command:

```
java -Dsolr.directoryFactory=HdfsDirectoryFactory
     -Dsolr.lock.type=hdfs
     -Dsolr.hdfs.home=hdfs://host:port/path
```

### The Block Cache

For performance, the HdfsDirectoryFactory uses a Directory that will cache HDFS blocks. This caching mechanism is meant to replace the standard file system cache that Solr utilizes so much. By default, this cache is allocated off heap. This cache will often need to be quite large and you may need to raise the off heap memory limit for the specific JVM you are running Solr in. For the Oracle/OpenJDK JVMs, the follow is an example command line parameter that you can use to raise the limit when starting Solr:

```
-XX:MaxDirectMemorySize=20g
```

### Settings

The HdfsDirectoryFactory has a number of settings.

#### Solr HDFS Settings

| Param | Example Value | Default | Description |
|---|---|---|---|
| `solr.hdfs.home` | hdfs://host:port/path/solr | N/A | A root location in HDFS for Solr to write collection data to. Rather than specifying an HDFS location for the data directory or update log directory, use this to specify one root location and have everything automatically created within this HDFS location. |

#### Block Cache Settings

| Param | Default | Description |
|---|---|---|
| `solr.hdfs.blockcache.enabled` | true | Enable the blockcache |
| `solr.hdfs.blockcache.read.enabled` | true | Enable the read cache |
| `solr.hdfs.blockcache.write.enabled` | true | Enable the write cache |
| `solr.hdfs.blockcache.direct.memory.allocation` | true | Enable direct memory allocation. If this is false, heap is used |
| `solr.hdfs.blockcache.slab.count` | 1 | Number of memory slabs to allocate. Each slab is 128 MB in size. |
| `solr.hdfs.blockcache.global` | false | Enable/Disable using one global cache for all SolrCores. The settings used will be from the first HdfsDirectoryFactory created. |

#### NRTCachingDirectory Settings

| Param | Default | Description |
|---|---|---|
| `solr.hdfs.nrtcachingdirectory.enable` | true | Enable the use of NRTCachingDirectory |
| `solr.hdfs.nrtcachingdirectory.maxmergesizemb` | 16 | NRTCachingDirectory max segment size for merges |
| `solr.hdfs.nrtcachingdirectory.maxcachedmb` | 192 | NRTCachingDirectory max cache size |

#### HDFS Client Configuration Settings

solr.hdfs.confdir pass the location of HDFS client configuration files - needed for HDFS HA for example.

| Param | Default | Description |
|---|---|---|
| `solr.hdfs.confdir` | N/A | Pass the location of HDFS client configuration files - needed for HDFS HA for example. |

#### Example

```
<directoryFactory name="DirectoryFactory" class="solr.HdfsDirectoryFactory">
  <str name="solr.hdfs.home">hdfs://host:port/solr</str>
  <bool name="solr.hdfs.blockcache.enabled">true</bool>
  <int name="solr.hdfs.blockcache.slab.count">1</int>
  <bool name="solr.hdfs.blockcache.direct.memory.allocation">true</bool>
  <int name="solr.hdfs.blockcache.blocksperbank">16384</int>
  <bool name="solr.hdfs.blockcache.read.enabled">true</bool>
  <bool name="solr.hdfs.blockcache.write.enabled">true</bool>
  <bool name="solr.hdfs.nrtcachingdirectory.enable">true</bool>
  <int name="solr.hdfs.nrtcachingdirectory.maxmergesizemb">16</int>
  <int name="solr.hdfs.nrtcachingdirectory.maxcachedmb">192</int>
</directoryFactory>
```

### Limitations

You must use an 'append-only' Lucene index codec because HDFS is an append only filesystem. The currently default codec used by Solr is 'append-only' and supported with HDFS.

### AutoAddReplica Settings

Collections created using autoAddReplica=true on a shared file system have auto addition of replica enabled. The following settings can be used to override the defaults in the solrcloud section of solr.xml.

| Param | Default | Description |
|---|---|---|
| autoReplicaFailoverWorkLoopDelay | 10000 | The time (in ms) between clusterstate inspections by the Overseer to detect and possibly act on creation of a replacement replica. |
| autoReplicaFailoverWaitAfterExpiration | 30000 | The minimum time (in ms) to wait for initiating replacement of a replica after first noticing it not being live. This is important to prevent false positives while stoping or starting the cluster. |
| autoReplicaFailoverBadNodeExpiration | 60000 | The delay (in ms) after which a replica marked as down would be unmarked. |

# SolrCloud

Apache Solr includes the ability to set up a cluster of Solr servers that combines fault tolerance and high availability. Called **SolrCloud**, these capabilities provide distributed indexing and search capabilities, supporting the following features:

- Central configuration for the entire cluster
- Automatic load balancing and fail-over for queries
- ZooKeeper integration for cluster coordination and configuration.

SolrCloud is flexible distributed search and indexing, without a master node to allocate nodes, shards and replicas. Instead, Solr uses ZooKeeper to manage these locations, depending on configuration files and schemas. Documents can be sent to any server and ZooKeeper will figure it out.

In this section, we'll cover everything you need to know about using Solr in SolrCloud mode. We've split up the details into the following topics:

- Getting Started with SolrCloud
- How SolrCloud Works
  - Shards and Indexing Data in SolrCloud
  - Distributed Requests
  - Read and Write Side Fault Tolerance
  - NRT, Replication, and Disaster Recovery with SolrCloud
- SolrCloud Configuration and Parameters
  - Using ZooKeeper to Manage Configuration Files
  - Collections API
  - Parameter Reference
  - Command Line Utilities
  - SolrCloud with Legacy Configuration Files

You can also find more information on the Solr wiki page on SolrCloud.

> ⚠ If upgrading an existing Solr 4.1 instance running with SolrCloud, be aware that the way the `name_node` parameter is defined has changed. This may cause a situation where the `name_node` uses the IP address of the machine instead of the server name, and thus SolrCloud is not aware of the existing node. If this happens, you can manually edit the `host` parameter in `solr.xml` to refer to the server name, or set the `host` in your system environment variables (since by default `solr.xml` is configured to inherit the `host` name from the environment variables). See also the section Solr Cores and solr.xml for more information about the `host` parameter.

## Getting Started with SolrCloud

SolrCloud is designed to provide a highly available, fault tolerant environment that can index your data for searching. It's a system in which data is organized into multiple pieces, or shards, that can be housed on multiple machines, with replicas providing redundancy for both scalability and fault tolerance, and a ZooKeeper server that helps manage the overall structure so that both indexing and search requests can be routed properly.

This section explains SolrCloud and its inner workings in detail, but before you dive in, it's best to have an idea of what it is you're trying to accomplish. This page provides a simple tutorial that explains how SolrCloud works on a practical level, and how to take advantage of its capabilities. We'll use simple examples of configuring SolrCloud on a single machine, which is obviously not a real production environment, which would include several servers or virtual machines. In a real production environment, you'll also use the real machine names instead of "localhost", which we've used here.

In this section you will learn:

- How to distribute data over multiple instances by using ZooKeeper and creating shards.
- How to create redundancy for shards by using replicas.
- How to create redundancy for the overall cluster by running multiple ZooKeeper instances.

Tutorials in this section:

This tutorial assumes that you're already familiar with the basics of using Solr. If you need a refresher, please visit the Getting Started section to get a grounding in Solr concepts. If you load documents as part of that exercise, you should start over with a fresh Solr installation for these SolrCloud tutorials.

### Interactive SolrCloud Example

The `bin/solr` script makes it easy to get started with SolrCloud as it walks you through the process of launching Solr nodes in cloud mode and adding a collection. To get started, simply do:

```
$ bin/solr -e cloud
```

This starts an interactive session to walk you through the steps of setting up a simple SolrCloud cluster with embedded ZooKeeper. The script starts by asking you how many Solr nodes you want to run in your local cluster, with the default being 2.

```
Welcome to the SolrCloud example!

This interactive session will help you launch a SolrCloud cluster on your local
workstation.
To begin, how many Solr nodes would you like to run in your local cluster? (specify
1-4 nodes) [2]
```

The script supports starting up to 4 nodes, but we recommend using the default of 2 when starting out. Next, the script will prompt you for the port to bind each of the Solr nodes to, such as:

```
 Please enter the port for node1 [8983]
```

Choose any available port for each node; the default for the first node is 8983 and 7574 for the second node. The script will start each node in order and shows you the command it uses to start the server, such as:

```
solr start -cloud -d node1 -p 8983
```

The first node will also start an embedded ZooKeeper server bound to port 9983.

After starting up all nodes in the cluster, the script prompts you for the name of the collection to create:

```
Please provide a name for your new collection: [gettingstarted]
```

The suggested default is "gettingstarted" but you should choose a better name for your specific search application. Next, the script prompts you for the number of shards to distribute the collection across. Sharding is covered in more detail later in this reference guide, so if you're unsure, we suggest using the default of 2 so that you can see how a collection is distributed across multiple nodes in a SolrCloud cluster.

Next, the script will prompt you for the number of replicas to create for each shard. Replication is covered in more detail later in the guide, so if you're unsure, then use the default of 2 so that you can see how replication is handled in SolrCloud. Lastly, the script will prompt you for the name of a configuration directory for your collection. You can choose **default** or **schemaless**. The default configuration directory is pulled from `example/solr/collection1/conf` and the schemaless configuration is pulled from the `example/example-schemaless/solr/collection1/conf` directory. The default configuration is more comprehensive and includes examples of most of Solr's core capabilities, whereas the schemaless configuration uses the field-guessing and managed schema features in Solr. The schemaless configuration is useful when you're still designing a schema for your documents and need some flexiblity as you experiment with Solr.

At this point, you should have a new collection created in your local SolrCloud cluster. To verify this, you can run the info command:

```
 $ bin/solr -i
```

You can see how your collection is deployed across the cluster by visiting the cloud panel in the Solr Admin UI: http://localhost:8983/solr/#/~cloud

You can restart your SolrCloud nodes using the `bin/solr` script. For instance, to restart node1 running on port 8983 (with an embedded ZooKeeper server), you would do:

```
$ bin/solr restart -c -p 8983 -d node1
```

To restart node2 running on port 7574, you can do:

```
$ bin/solr restart -c -p 7574 -d node2 -z localhost:9983
```

Notice that you need to specify the ZooKeeper address (-z localhost:9983) when starting node2 so that it can join the cluster with node1.

The next section walks you through the manual process of setting up a SolrCloud cluster instead of using the `bin/solr` script (which performed all the manual steps for you).

### Simple Two-Shard Cluster on the Same Machine

Creating a cluster with multiple shards involves two steps:

1. Start the first node, which will include an embedded ZooKeeper server to keep track of your cluster.
2. Start any remaining shard nodes and point them to the running ZooKeeper.

> Make sure to run Solr from the example directory in non-SolrCloud mode at least once before beginning; this process unpacks the jar files necessary to run SolrCloud. However, do not load documents yet, just start it once and shut it down.

In this example, you'll create two separate Solr instances on the same machine. This is not a production-ready installation, but just a quick exercise to get you familiar with SolrCloud.

For this exercise, we'll start by creating two copies of the `example` directory that is part of the Solr distribution:

```
cd <SOLR_DIST_HOME>
cp -r example node1
cp -r example node2
```

These copies of the `example` directory can really be called anything. All we're trying to do is copy Solr's example app to the side so we can play with it and still have a stand-alone Solr example to work with later if we want.

Next, start the first Solr instance, including the `-DzkRun` parameter, which also starts a local ZooKeeper instance:

```
cd node1
java -DzkRun -DnumShards=2 -Dbootstrap_confdir=./solr/collection1/conf
-Dcollection.configName=myconf -jar start.jar
```

Let's look at each of these parameters:

**`-DzkRun`** Starts up a ZooKeeper server embedded within Solr. This server will manage the cluster configuration. Note that we're doing this example all on one machine; when you start working with a production system, you'll likely use multiple ZooKeepers in an ensemble (or at least a stand-alone ZooKeeper instance). In that case, you'll replace this parameter with `zkHost=<ZooKeeper Host:Port>`, which is the hostname:port of the stand-alone ZooKeeper.

**`-DnumShards`** Determines how many pieces you're going to break your index into. In this case we're going to break the index into two pieces, or *shards*, so we're setting this value to 2. The default value, if not specified, is 1.

**`-Dbootstrap_confdir`** ZooKeeper needs to get a copy of the cluster configuration, so this parameter tells it where to find that information.

**`-Dcollection.configName`** This parameter determines the name under which that configuration information is stored by ZooKeeper. We've used "myconf" as an example, it can be anything you'd like.

> The `-DnumShards`, `-Dbootstrap_confdir`, and `-Dcollection.configName` parameters need only be specified once, the first time you start Solr in SolrCloud mode. They load your configurations into ZooKeeper; if you run them again at a later time, they will re-load your configurations and may wipe out changes you have made.

At this point you have one sever running, but it represents only half the shards, so you will need to start the second one before you have a fully functional cluster. To do that, start the second instance in another window as follows:

```
cd node2
java -Djetty.port=7574 -DzkHost=localhost:9983 -jar start.jar
```

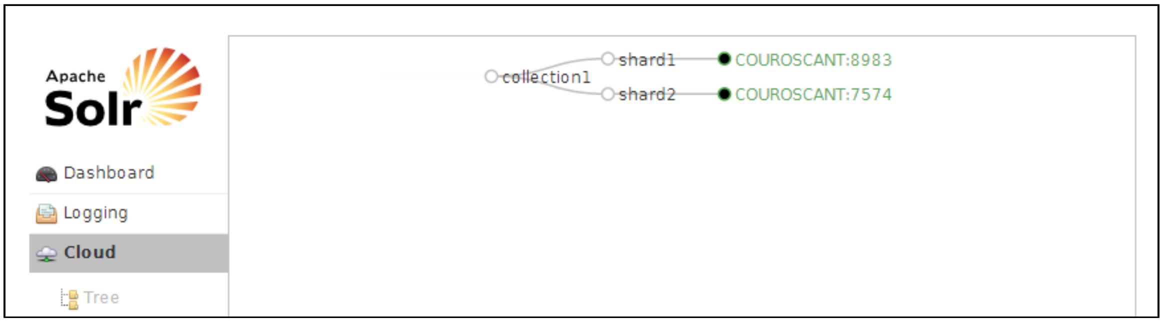
Because this node isn't running ZooKeeper, and didn't involve bootstraping collection1, the parameters are a bit less complex:

**`-Djetty.port`** The only reason we even have to set this parameter is because we're running both servers on the same machine, so they can't both use Jetty's default port. In this case we're choosing an arbitrary number that's different from the default. When you start on different machines, you can use the same Jetty ports if you'd like.

**`-DzkHost`** This parameter tells Solr where to find the ZooKeeper server so that it can "report for duty". By default, the ZooKeeper server operates on the Solr port plus 1000. (Note that if you were running an external ZooKeeper server, you'd simply point to that.)

At this point you should have two Solr windows running, both being managed by ZooKeeper. To verify that, open the Solr Admin UI in your browser and go to the Cloud screen of the first Solr server you started: http://localhost:8983/solr/#/~cloud

You should see both node1 and node2, as in:



Now it's time to see the cluster in action. Start by indexing some data to one or both shards. You can do this any way you like, but the easiest way is to use the `exampledocs`, along with curl so that you can control which port (and thereby which server) gets the updates:

```
curl http://localhost:8983/solr/update?commit=true -H "Content-Type: text/xml" -d
"@mem.xml"
curl http://localhost:7574/solr/update?commit=true -H "Content-Type: text/xml" -d
"@monitor2.xml"
```

At this point each shard contains a subset of the data, but a search directed at either server should span both shards. For example, the following searches should both return the identical set of all results:

`http://localhost:8983/solr/collection1/select?q=*:*`

`http://localhost:7574/solr/collection1/select?q=*:*`

The reason that this works is that each shard knows about the other shards, so the search is carried out on all cores, then the results are combined and returned by the called server.

In this way you can have two cores or two hundred, with each containing a separate portion of the data.

If you want to check the number of documents on each shard, you could add `distrib=false` to each query and your search would not span all shards.

But what about providing high availability, even if one of these servers goes down? To do that, you'll need to look at replicas.

### Two-Shard Cluster with Replicas

In order to provide high availability, you can create replicas, or copies of each shard that run in parallel with the main core for that shard. The architecture consists of the original shards, which are called the leaders, and their replicas, which contain the same data but let the leader handle all of the administrative tasks such as making sure data goes to all of the places it should go. This way, if one copy of the shard goes down, the data is still available and the cluster can continue to function.

Start by creating two more fresh copies of the example directory:

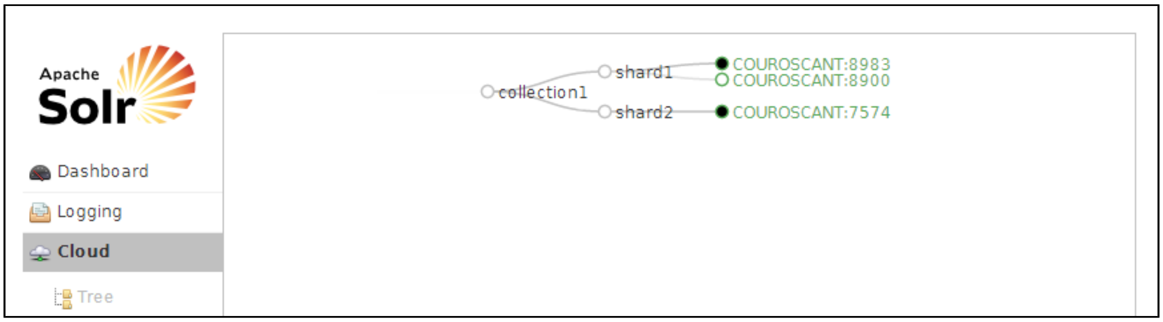
```
cd <SOLR_DIST_HOME>
cp -r example node3
cp -r example node4
```

Just as when we created the first two shards, you can name these copied directories whatever you want.

If you don't already have the two instances you created in the previous section up and running, go ahead and restart them. From there, it's simply a matter of adding additional instances. Start by adding node3:

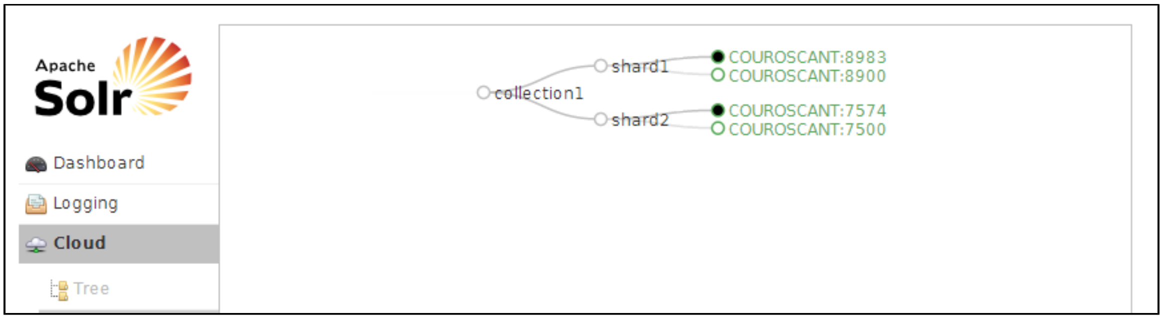
```
cd node3
java -Djetty.port=8900 -DzkHost=localhost:9983 -jar start.jar
```

Notice that the parameters are exactly the same as they were for starting the second node; you're simply pointing a new instance at the original ZooKeeper. But if you look at the SolrCloud admin page, you'll see that it was added not as a third shard, but as a replica for the first:



This is because the cluster already knew that there were only two shards and they were already accounted for, so new nodes are added as replicas. Similarly, when you add the fourth instance, it's added as a replica for the second shard:

```
cd node4
java -Djetty.port=7500 -DzkHost=localhost:9983 -jar start.jar
```



If you were to add additional instances, the cluster would continue this round-robin, adding replicas as necessary. Replicas are attached to leaders in the order in which they are started, unless they are assigned to a specific shard with an additional parameter of `shardId` (as a system property, as in `-DshardId=1`, the value of which is the ID number of the shard the new node should be attached to). Upon restarts, the node will still be attached to the same leader even if the `shardId` is not defined again (it will always be attached to that machine).

So where are we now? You now have four servers to handle your data. If you were to send data to a replica, as in:

```
curl http://localhost:7500/solr/update?commit=true -H "Content-Type: text/xml" -d
"@money.xml"
```

the course of events goes like this:

1. Replica (in this case the server on port 7500) gets the request.
2. Replica forwards request to its leader (in this case the server on port 7574).
3. The leader processes the request, and makes sure that all of its replicas process the request as well.

In this way, the data is available via a request to any of the running instances, as you can see by requests to:

`http://localhost:8983/solr/collection1/select?q=*:*`

`http://localhost:7574/solr/collection1/select?q=*:*`

`http://localhost:8900/solr/collection1/select?q=*:*`

`http://localhost:7500/solr/collection1/select?q=*:*`

But how does this help provide high availability? Simply put, a cluster must have at least one server running for each shard in order to function. To test this, shut down the server on port 7574, and then check the other servers:

`http://localhost:8983/solr/collection1/select?q=*:*`

`http://localhost:8900/solr/collection1/select?q=*:*`

`http://localhost:7500/solr/collection1/select?q=*:*`

You should continue to see the full set of data, even though one of the servers is missing. In fact, you can have multiple servers down, and as long as at least one instance for each shard is running, the cluster will continue to function. If the leader goes down – as in this example – a new leader will be "elected" from among the remaining replicas.

Note that when we talk about servers going down, in this example it's crucial that one particular server stays up, and that's the one running on port 8983. That's because it's the instance running ZooKeeper. If that goes down, the cluster can continue to function under some circumstances, but it won't be able to adapt to any servers that come up or go down.

That kind of single point of failure is obviously unacceptable. Fortunately, there is a solution for this problem: multiple ZooKeepers.

### Using Multiple ZooKeepers in an Ensemble

> To simplify setup for this example we're using the internal ZooKeeper server that comes with Solr, but in a production environment, you will likely be using an external ZooKeeper. The concepts are the same, however. You can find instructions on setting up an external ZooKeeper server here: http://zookeeper.apache.org/doc/r3.3.4/zookeeperStarted.html

To truly provide high availability, we need to make sure that not only do we also have at least one shard server running at all times, but also that the cluster also has a ZooKeeper running to manage it. To do that, you can set up a cluster to use multiple ZooKeepers. This is called using a ZooKeeper ensemble.

A ZooKeeper ensemble can keep running as long as more than half of its servers are up and running, so at least two servers in a three ZooKeeper ensemble, 3 servers in a 5 server ensemble, and so on, must be running at any given time. These required servers are called a quorum.

In this example, you're going to set up the same two-shard cluster you were using before, but instead of a single ZooKeeper, you'll run a ZooKeeper server on three of the instances. Start by cleaning up any ZooKeeper data from the previous example:

```
cd <SOLR_DIST_DIR>
rm -r node*/solr/zoo_data
```

Next you're going to restart the Solr servers, but this time, rather than having them all point to a single ZooKeeper instance, each will run ZooKeeper **and** listen to the rest of the ensemble for instructions.

You're using the same ports as before – 8983, 7574, 8900 and 7500 – so any ZooKeeper instances would run on ports 9983, 8574, 9900 and 8500. You don't actually need to run ZooKeeper on every single instance, however, so assuming you run ZooKeeper on 9983, 8574, and 9900, the ensemble would have an address of:

```
localhost:9983,localhost:8574,localhost:9900
```

This means that when you start the first instance, you'll do it like this:

```
cd node1
java -DzkRun -DnumShards=2 -Dbootstrap_confdir=./solr/collection1/conf \
   -Dcollection.configName=myconf \
   -DzkHost=localhost:9983,localhost:8574,localhost:9900 \
   -jar start.jar
```

> Note that the order of the parameters matters. Make sure to specify the -DzkHost parameter after the other ZooKeeper-related parameters.

You'll notice a lot of error messages scrolling past; this is because the ensemble doesn't yet have a quorum of ZooKeepers running.

Notice also, that this step takes care of uploading the cluster's configuration information to ZooKeeper, so starting the next server is more straightforward:

```
cd node2
java -Djetty.port=7574 -DzkRun -DnumShards=2 \
    -DzkHost=localhost:9983,localhost:8574,localhost:9900 -jar start.jar
```

Once you start this instance, you should see the errors begin to disappear on both instances, as the ZooKeepers begin to update each other, even though you only have two of the three ZooKeepers in the ensemble running.

Next start the last ZooKeeper:

```
cd node3
java -Djetty.port=8900 -DzkRun -DnumShards=2 \
    -DzkHost=localhost:9983,localhost:8574,localhost:9900 -jar start.jar
```

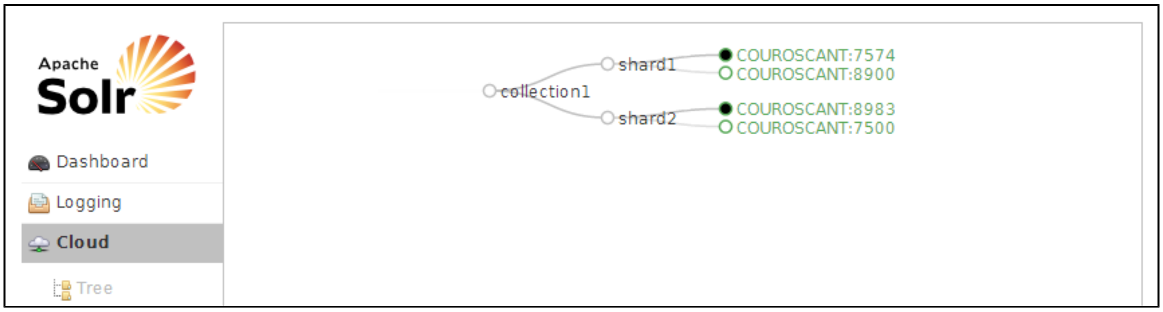
Finally, start the last replica, which doesn't itself run ZooKeeper, but references the ensemble:

```
cd node4
java -Djetty.port=7500 -DzkHost=localhost:9983,localhost:8574,localhost:9900 \
    -jar start.jar
```

Just to make sure everything's working properly, run a query:

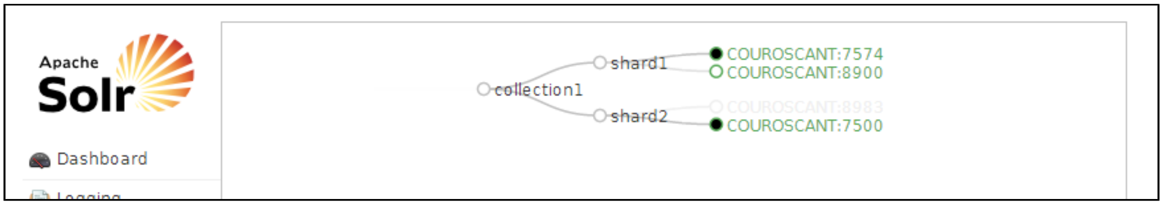
`http://localhost:8983/solr/collection1/select?q=*:*`

and check the SolrCloud admin page:



Now you can go ahead and kill the server on 8983, but ZooKeeper will still work, because you have more than half of the original servers still running. To verify, open the SolrCloud admin page on another server, such as:

http://localhost:8900/solr/#/~cloud



## How SolrCloud Works

In this section, we'll discuss generally how SolrCloud works, covering these topics:

- Nodes, Cores, Clusters and Leaders
- Shards and Indexing Data in SolrCloud
- Distributed Requests
- Read and Write Side Fault Tolerance

- NRT, Replication, and Disaster Recovery with SolrCloud

If you are already familiar with SolrCloud concepts and functionality, you can skip to the section covering SolrCloud Configuration and Parameters.

### Basic SolrCloud Concepts

On a single node, Solr has a **core** that is essentially a single **index**. If you want multiple indexes, you create multiple cores. With SolrCloud, a single index can span multiple Solr instances. This means that a single index can be made up of multiple cores on different machines.

The cores that make up one logical index are called a **collection**. A collection is a essentially a single index that can span many cores, both for index scaling as well as redundancy. If, for instance, you wanted to move your two-core Solr setup to SolrCloud, you would have 2 collections, each made up of multiple individual cores.

In SolrCloud you can have multiple collections. Collections can be divided into slices. Each slice can exist in multiple copies; these copies of the same slice are called **shards**. One of the shards within a slice is the **leader**, designated by a leader-election process. Each shard is a physical index, so one shard corresponds to one core.

It is important to understand the distinction between a core and a collection. In classic single node Solr, a core is basically equivalent to a collection in that it presents one logical index. In SolrCloud, the cores on multiple nodes form a collection. This is still just one logical index, but multiple cores host different shards of the full collection. So a core encapsulates a single physical index on an instance. A collection is a combination of all of the cores that together provide a logical index that is distributed across many nodes.

### Differences Between Solr 3.x-style Scaling and SolrCloud

In Solr 3.x, Solr included following features:

- The index and all changes to it are replicated to another Solr instance.
- In distributed searches, queries are sent to multiple Solr instances and the results are combined into a single output.
- Documents are available only after committing, which may be expensive and not very timely.
- Sharding must be done manually, usually through SolrJ or a similar utility, and there is no distributed indexing: your index code must understand your sharding schema.
- Replication must be manually configured and can slow down access to recent content because the system needs to wait for a commit and the replication to be triggered and to complete.
- Failure recovery may result in the loss of your ability to index, and make recovering your indexing process difficult.

With SolrCloud, some capabilities are distributed:

- SolrCloud automatically distributes index updates to the appropriate shard, distributes searches across multiple shards, and assigns replicas to shards when replicas are available.
- Near Real Time searching is supported, and if configured, documents are available after a "soft" commit.
- Indexing accesses your sharding schema automatically.
- Replication is automatic for backup purposes.
- Recovery is robust and automatic.
- ZooKeeper serves as a repository for cluster state.

## Nodes, Cores, Clusters and Leaders

#### Nodes and Cores

In SolrCloud, a *node* is Java Virtual Machine instance running Solr, commonly called a server. Each Solr core can also be considered a node. Any node can contain both an instance of Solr and various kinds of data.

A Solr *core* is basically an index of the text and fields found in documents. A single Solr instance can contain multiple "cores", which are separate from each other based on local criteria. It might be that they are going to provide different search interfaces to users (customers in the US and customers in Canada, for example), or they have security concerns (some users cannot have access to some documents), or the documents are really different and just won't mix well in the same index (a shoe database and a dvd database).

When you start a new core in SolrCloud mode, it registers itself with ZooKeeper. This involves creating an Ephemeral node that will go away if the Solr instance goes down, as well as registering information about the core and how to contact it (such as the base Solr URL, core name, etc). Smart clients and nodes in the cluster can use this information to determine who they need to talk to in order to fulfill a request.

New Solr cores may also be created and associated with a collection via CoreAdmin. Additional cloud-related parameters are discussed in the Parameter Reference page. Terms used for the CREATE action are:

- **collection**: the name of the collection to which this core belongs. Default is the name of the core.
- **shard**: the shard id this core represents. (Optional: normally you want to be auto assigned a shard id.)
- **collection.<param>=<value>**: causes a property of `<param>=<value>` to be set if a new collection is being created. For example, use `collection.configName=<configname>` to point to the config for a new collection.

For example:

```
curl  'http://localhost:8983/solr/admin/cores?
     action=CREATE&name=mycore&collection=collection1&shard=shard2'
```

### Clusters

A cluster is set of Solr nodes managed by ZooKeeper as a single unit. When you have a cluster, you can always make requests to the cluster and if the request is acknowledged, you can be sure that it will be managed as a unit and be durable, i.e., you won't lose data. Updates can be seen right after they are made and the cluster can be expanded or contracted.

#### *Creating a Cluster*

A cluster is created as soon as you have more than one Solr instance registered with ZooKeeper. The section Getting Started with SolrCloud reviews how to set up a simple cluster.

#### *Resizing a Cluster*

Clusters contain a settable number of shards. You set the number of shards for a new cluster by passing a system property, `numShards`, when you start up Solr. The `numShards` parameter must be passed on the first startup of any Solr node, and is used to auto-assign which shard each instance should be part of. Once you have started up more Solr nodes than `numShards`, the nodes will create replicas for each shard, distributing them evenly across the node, as long as they all belong to the same collection.

To add more cores to your collection, simply start the new core. You can do this at any time and the new core will sync its data with the current replicas in the shard before becoming active.

You can also avoid `numShards` and manually assign a core a shard ID if you choose.

The number of shards determines how the data in your index is broken up, so you cannot change the number of shards of the index after initially setting up the cluster.

However, you do have the option of breaking your index into multiple shards to start with, even if you are only using a single machine. You can then expand to multiple machines later. To do that, follow these steps:

1. Set up your collection by hosting multiple cores on a single physical machine (or group of machines). Each of these shards will be a leader for that shard.
2. When you're ready, you can migrate shards onto new machines by starting up a new replica for a given shard on each new machine.
3. Remove the shard from the original machine. ZooKeeper will promote the replica to the leader for that shard.

#### Leaders and Replicas

The concept of a *leader* is similar to that of *master* when thinking of traditional Solr replication. The leader is responsible for making sure the *replicas* are up to date with the same information stored in the leader.

However, with SolrCloud, you don't simply have one master and one or more "slaves", instead you likely have distributed your search and index traffic to multiple machines. If you have bootstrapped Solr with `numShards=2`, for example, your indexes are split across both shards. In this case, both shards are considered leaders. If you start more Solr nodes after the initial two, these will be automatically assigned as replicas for the leaders.

Replicas are assigned to shards in the order they are started the first time they join the cluster. This is done in a round-robin manner, unless the new node is manually assigned to a shard with the `shardId` parameter during startup. This parameter is used as a system property, as in `-DshardId=1`, the value of which is the ID number of the shard the new node should be attached to.

On subsequent restarts, each node joins the same shard that it was assigned to the first time the node was started (whether that assignment happened manually or automatically). A node that was previously a replica, however, may become the leader if the previously assigned leader is not available.

Consider this example:

- Node A is started with the bootstrap parameters, pointing to a stand-alone ZooKeeper, with the `numShards` parameter set to 2.
- Node B is started and pointed to the stand-alone ZooKeeper.

Nodes A and B are both shards, and have fulfilled the 2 shard slots we defined when we started Node A. If we look in the Solr Admin UI, we'll see that both nodes are considered leaders (indicated with a solid blank circle).

- Node C is started and pointed to the stand-alone ZooKeeper.

Node C will automatically become a replica of Node A because we didn't specify any other shard for it to belong to, and it cannot become a new shard because we only defined two shards and those have both been taken.

- Node D is started and pointed to the stand-alone ZooKeeper.

Node D will automatically become a replica of Node B, for the same reasons why Node C is a replica of Node A.

Upon restart, suppose that Node C starts before Node A. What happens? Node C will become the leader, while Node A becomes a replica of Node C.

### Shards and Indexing Data in SolrCloud

When your data is too large for one node, you can break it up and store it in sections by creating one or more **shards**. Each is a portion of the logical index, or core, and it's the set of all nodes containing that section of the index.

A shard is a way of splitting a core over a number of "servers", or nodes. For example, you might have a shard for data that represents each state, or different categories that are likely to be searched independently, but are often combined.

Before SolrCloud, Solr supported Distributed Search, which allowed one query to be executed across multiple shards, so the query was executed against the entire Solr index and no documents would be missed from the search results. So splitting the core across shards is not exclusively a SolrCloud concept. There were, however, several problems with the distributed approach that necessitated improvement with SolrCloud:

1. Splitting of the core into shards was somewhat manual.
2. There was no support for distributed indexing, which meant that you needed to explicitly send documents to a specific shard; Solr couldn't figure out on its own what shards to send documents to.
3. There was no load balancing or failover, so if you got a high number of queries, you needed to figure out where to send them and if one shard died it was just gone.

SolrCloud fixes all those problems. There is support for distributing both the index process and the queries automatically, and ZooKeeper provides failover and load balancing. Additionally, every shard can also have multiple replicas for additional robustness.

Unlike Solr 3.x, in SolrCloud there are no masters or slaves. Instead, there are leaders and replicas. Leaders are automatically elected, initially on a first-come-first-served basis, and then based on the Zookeeper process described at http://zookeeper.apache.org/doc/trunk/recipes.html#sc_leaderElection..

If a leader goes down, one of its replicas is automatically elected as the new leader. As each node is started, it's assigned to the shard with the fewest replicas. When there's a tie, it's assigned to the shard with the lowest shard ID.

When a document is sent to a machine for indexing, the system first determines if the machine is a replica or a leader.

- If the machine is a replica, the document is forwarded to the leader for processing.
- If the machine is a leader, SolrCloud determines which shard the document should go to, forwards the document the leader for that shard, indexes the document for this shard, and forwards the index notation to itself and any replicas.

### Document Routing

Solr offers the ability to specify the router implementation used by a collection by specifying the `router.name` parameter when creating your collection. If you use the "`compositeId`" router, you can send documents with a prefix in the document ID which will be used to calculate the hash Solr uses to determine the shard a document is sent to for indexing. The prefix can be anything you'd like it to be (it doesn't have to be the shard name, for example), but it must be consistent so Solr behaves consistently. For example, if you wanted to co-locate documents for a customer, you could use the customer name or ID as the prefix. If your customer is "IBM", for example, with a document with the ID "12345", you would insert the prefix into the document id field: "IBM!12345". The exclamation mark ('!') is critical here, as it distinguishes the prefix used to determine which shard to direct the document to.

Then at query time, you include the prefix(es) into your query with the `_route_` parameter (i.e., `q=solr&_route_=IBM!`) to direct queries to specific shards. In some situations, this may improve query performance because it overcomes network latency when querying all the shards.

> ⚠ The `_route_` parameter replaces `shard.keys`, which has been deprecated and will be removed in a future Solr release.

The `compositeId` router supports prefixes containing up to 2 levels of routing. For example: a prefix routing first by region, then by customer: "USA!IBM!12345"

If you do not want to influence how documents are stored, you don't need to specify a prefix in your document ID.

If you created the collection and defined the "implicit" router at the time of creation, you can additionally define a `router.field` parameter to use a field from each document to identify a shard where the document belongs. If the field specified is missing in the document, however, the document will be rejected. You could also use the `_route_` parameter to name a specific shard.

### Shard Splitting

Until Solr 4.3, when you created a collection in SolrCloud, you had to decide on your number of shards when you created the collection and you could not change it later. It can be difficult to know in advance the number of shards that you need, particularly when organizational requirements can change at a moment's notice, and the cost of finding out later that you chose wrong can be high, involving creating new cores and re-indexing all of your data.

The ability to split shards is in the Collections API. It currently allows splitting a shard into two pieces. The existing shard is left as-is, so the split action effectively makes two copies of the data as new shards. You can delete the old shard at a later time when you're ready.

More details on how to use shard splitting is in the section on the Collections API.

## Distributed Requests

One of the advantages of using SolrCloud is the ability to distribute requests among various shards that may or may not contain the data that you're looking for.  You have the option of searching over all of your data or just parts of it.

Querying all shards for a collection should look familiar; it's as though SolrCloud didn't even come into play:

```
http://localhost:8983/solr/collection1/select?q=*:*
```

If, on the other hand, you wanted to search just one shard, you can specify that shard, as in:

```
http://localhost:8983/solr/collection1/select?q=*:*&shards=localhost:7574/solr
```

If you want to search a group of shards, you can specify them together:

```
http://localhost:8983/solr/collection1/select?q=*:*&shards=localhost:7574/solr,localhost:8983/solr
```

Or you can specify a list of servers to choose from for load balancing purposes by using the pipe symbol (|):

```
http://localhost:8983/solr/collection1/select?q=*:*&shards=localhost:7574/solr|localhost:7500/solr
```

(If you have explicitly created your shards using ZooKeeper and have shard IDs, you can use those IDs rather than server addresses.)

You also have the option of searching multiple collections. For example:

```
http://localhost:8983/solr/collection1/select?collection=collection1,collection2,collection3
```

## Read and Write Side Fault Tolerance

#### Read Side Fault Tolerance

With earlier versions of Solr, you had to set up your own load balancer. Now each individual node load balances requests across the replicas in a cluster. You still need a load balancer on the 'outside' that talks to the cluster, or you need a smart client (Solr provides a smart Java Solrj client called CloudSolrServer).

A smart client understands how to read and interact with ZooKeeper and only requests the ZooKeeper ensemble's address to start discovering to which nodes it should send requests.

Each distributed search request is executed against all shards for a collection unless limited by the user with the '`shards`' or '_route_' parameters. If one or more shards queried are unavailable then the default is to fail the request. However, there are many use-cases where partial results are acceptable and so Solr provides a boolean `shards.tolerant` parameter (default '`false`'). If `shards.tolerant=true` then partial results may be returned. If the returned response does not contain results from all the appropriate shards then the response header contains a special flag called '`partialResults`'. The client can specify '`shards.info`' along with the '`shards.tolerant`' parameter to retrieve more fine-grained details.

Example response with `partialResults` flag set to 'true':

**Solr Response with partialResults**

```
{
  "responseHeader": {
    "status": 0,
    "partialResults": true,
    "QTime": 20,
    "params": {
      "wt": "json"
    }
  },
  "response": {
    "numFound": 77,
    "start": 0,
    "docs": [  ]
  }
}
```

#### Write Side Fault Tolerance

SolrCloud supports near real-time actions, elasticity, high availability, and fault tolerance. What this means, basically, is that when you have a large cluster, you can always make requests to the cluster, and if a request is acknowledged you are sure it will be durable; i.e., you won't lose data. Updates can be seen right after they are made and the cluster can be expanded or contracted.

##### *Recovery*

A Transaction Log is created for each node so that every change to content or organization is noted.  The log is used to determine which content in the node should be included in a replica. When a new replica is created, it refers to the Leader and the Transaction Log to know which content to include. If it fails, it retries.

Since the Transaction Log consists of a record of updates, it allows for more robust indexing because it includes redoing the uncommitted updates if indexing is interrupted.

If a leader goes down, it may have sent requests to some replicas and not others. So when a new potential leader is identified, it runs a synch process against the other replicas.  If this is successful, everything should be consistent, the leader registers as active, and normal actions proceed. If the a replica is too far out of synch, the system asks for a full replication/replay-based recovery.

If an update fails because cores are reloading schemas and some have finished but others have not, the leader tells the nodes that the update failed and starts the recovery procedure.

#### *Achieved Replication Factor*

When using a replication factor greater than one, an update request may succeed on the shard leader but fail on one or more of the replicas. For instance, consider a collection with one shard and replication factor of three. In this case, you have a shard leader and two additional replicas. If an update request succeeds on the leader but fails on both replicas, for whatever reason, the update request is still considered successful from the perspective of the client. The replicas that missed the update will sync with the leader when they recover.

Behind the scenes, this means that Solr has accepted updates that are only on one of the nodes (the current leader). Solr supports the optional `min_rf` parameter on update requests that cause the server to return the achieved replication factor for an update request in the response. For the example scenario described above, if the client application included min_rf >= 1, then Solr would return rf=1 in the Solr response header because the request only succeeded on the leader. The update request will still be accepted as the `min_rf` parameter only tells Solr that the client application wishes to know what the achieved replication factor was for the update request. In other words, min_rf does not mean Solr will enforce a minimum replication factor as Solr does not support rolling back updates that succeed on a subset of replicas.

On the client side, if the achieved replication factor is less than the acceptable level, then the client application can take additional measures to handle the degraded state. For instance, a client application may want to keep a log of which update requests were sent while the state of the collection was degraded and then resend the updates once the problem has been resolved. In short, `min_rf` is an optional mechanism for a client application to be warned that an update request was accepted while the collection is in a degraded state.

## NRT, Replication, and Disaster Recovery with SolrCloud

### SolrCloud and Replication

Replication ensures redundancy for your data, and enables you to send an update request to any node in the shard.  If that node is a replica, it will forward the request to the leader, which then forwards it to all existing replicas, using versioning to make sure every replica has the most up-to-date version.  This architecture enables you to be certain that your data can be recovered in the event of a disaster, even if you are using Near Real Time searching.

### Near Real Time Searching

If you want to use the NearRealtimeSearch support, enable auto soft commits in your `solrconfig.xml` file before storing it into Zookeeper. Otherwise you can send explicit soft commits to the cluster as you need.

SolrCloud doesn't work very well with separated data clusters connected by an expensive pipe. The root problem is that SolrCloud's architecture sends documents to all the nodes in the cluster (on a per-shard basis), and that architecture is really dictated by the NRT functionality.

Imagine that you have a set of servers in China and one in the US that are aware of each other. Assuming 5 replicas, a single update to a shard may make multiple trips over the expensive pipe before it's all done, probably slowing indexing speed unacceptably.

So the SolrCloud recommendation for this situation is to maintain these clusters separately; nodes in China don't even know that nodes exist in the US and vice-versa.  When indexing, you send the update request to one node in the US and one in China and all the node-routing after that is local to the separate clusters. Requests can go to any node in either country and maintain a consistent view of the data.

However, if your US cluster goes down, you have to re-synchronize the down cluster with up-to-date information from China. The process requires you to replicate the index from China to the repaired US installation and then get everything back up and working.

#### Disaster Recovery for an NRT system

Use of Near Real Time (NRT) searching affects the way that systems using SolrCloud behave during disaster recovery.

The procedure outlined below assumes that you are maintaining separate clusters, as described above.   Consider, for example, an event in which the US cluster goes down (say, because of a hurricane), but the China cluster is intact.  Disaster recovery consists of creating the new system and letting the intact cluster create a replicate for each shard on it, then promoting those replicas to be leaders of the newly created US cluster.

Here are the steps to take:

1. Take the downed system offline to all end users.
2. Take the indexing process offline.
3. Repair the system.
4. Bring up one machine per shard in the repaired system as part of the ZooKeeper cluster on the good system, and wait for replication to happen, creating a replica on that machine.  (SoftCommits will not be repeated, but data will be pulled from the transaction logs if necessary.)

   > SolrCloud will automatically use old-style replication for the bulk load. By temporarily having only one replica, you'll minimize data transfer across a slow connection.

5. Bring the machines of the repaired cluster down, and reconfigure them to be a separate Zookeeper cluster again, optionally adding more replicas for each shard.
6. Make the repaired system visible to end users again.
7. Start the indexing program again, delivering updates to both systems.

## SolrCloud Configuration and Parameters

In this section, we'll cover the various configuration options for SolrCloud.

In general, with a new Solr 4 instance, the required configuration is in the sample `schema.xml` and `solrconfig.xml` files. However, there may be reasons to change default settings or configure the cloud elements manually.

The following sections cover these topics:

- Setting Up an External ZooKeeper Ensemble
- Using ZooKeeper to Manage Configuration Files
- Collections API
- Parameter Reference
- Command Line Utilities
- SolrCloud with Legacy Configuration Files

### Setting Up an External ZooKeeper Ensemble

Although Solr comes bundled with Apache ZooKeeper, you should consider yourself discouraged from using this internal ZooKeeper in production, because shutting down a redundant Solr instance will also shut down its ZooKeeper server, which might not be quite so redundant. Because a ZooKeeper ensemble must have a quorum of more than half its servers running at any given time, this can be a problem.

The solution to this problem is to set up an external ZooKeeper ensemble. Fortunately, while this process can seem intimidating due to the number of powerful options, setting up a simple ensemble is actually quite straightforward. The basic steps are as follows:

#### Download Apache ZooKeeper

The first step in setting up Apache ZooKeeper is, of course, to download the software. It's available from http://zookeeper.apache.org/releases.html.

> ⚠ When using stand-alone ZooKeeper, you need to take care to keep your version of ZooKeeper updated with the latest version distributed with Solr. Since you are using it as a stand-alone application, it does not get upgraded when you upgrade Solr.
>
> Solr 4.0 uses Apache ZooKeeper v3.3.6.
>
> Solr 4.1 through 4.7 use Apache ZooKeeper v3.4.5.
>
> Solr 4.8 and higher uses Apache ZooKeeper v3.4.6.

#### Setting Up a Single ZooKeeper

##### *Create the instance*

Creating the instance is a simple matter of extracting the files into a specific target directory. The actual directory itself doesn't matter, as long as you know where it is, and where you'd like to have ZooKeeper store its internal data.

##### *Configure the instance*

The next step is to configure your ZooKeeper instance. To do that, create the following file: `<ZOOKEEPER_HOME>/conf/zoo.cfg`. To this file, add the following information:

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
```

The parameters are as follows:

**tickTime**: Part of what ZooKeeper does is to determine which servers are up and running at any given time, and the minimum session time out is defined as two "ticks". The `tickTime` parameter specifies, in miliseconds, how long each tick should be.

**dataDir**: This is the directory in which ZooKeeper will store data about the cluster. This directory should start out empty.

**clientPort**: This is the port on which Solr will access ZooKeeper.

Once this file is in place, you're ready to start the ZooKeeper instance.

##### *Run the instance*

To run the instance, you can simply use the `ZOOKEEPER_HOME/bin/zkServer.sh` script provided, as with this command: `zkServer.sh start`

Again, ZooKeeper provides a great deal of power through additional configurations, but delving into them is beyond the scope of this tutorial. For more information, see the ZooKeeper Getting Started page. For this example, however, the defaults are fine.

#### *Point Solr at the instance*

Pointing Solr at the ZooKeeper instance you've created is a simple matter of using the `-DzkHost` parameter. For example, in the Getting Started with SolrCloud example you learned how to point to the internal ZooKeeper. In this example, you would point to the ZooKeeper you've started on port 2181.

On the first server:

```
cd shard1
java -DnumShards=2 -Dbootstrap_confdir=./solr/collection1/conf
-Dcollection.configName=myconf -DzkHost=localhost:2181 -jar start.jar
```

On each subsequent server:

```
cd shard2java
java -Djetty.port=7574 -DzkHost=localhost:2181 -jar start.jar
```

As with the Getting Started with SolrCloud example, you must first upload the configuration information, and then you can connect a second, third, etc., instance.

#### *Shut down ZooKeeper*

To shut down ZooKeeper, use the zkServer script with the "stop" command: `zkServer.sh stop`.

### Setting up a ZooKeeper Ensemble

In the Getting Started example, using a ZooKeeper ensemble was a simple matter of starting multiple instances and pointing to them. With an external ZooKeeper ensemble, you need to set things up just a little more carefully.

The difference is that rather than simply starting up the servers, you need to configure them to know about and talk to each other first. So your original `zoo.cfg` file might look like this:

```
dataDir=/var/lib/zookeeperdata/1
clientPort=2181
initLimit=5
syncLimit=2
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Here you see three new parameters:

**initLimit**: The time, in ticks, the server allows for connecting to the leader. In this case, you have 5 ticks, each of which is 2000 milliseconds long, so the server will wait as long as 10 seconds to connect.

**syncLimit**: The time, in ticks, the server will wait before updating itself from the leader.

**server.X**: These are the IDs and locations of all servers in the ensemble, the ports on which they communicate with each other. The server ID must additionally stored in the `<dataDir>/myid` file and be located in the `dataDir` of each ZooKeeper instance. The ID identifies each server, so in the case of this first instance, you would create the file `/var/lib/zookeeperdata/1/myid` with the content "1".

Now, whereas with Solr you need to create entirely new directories to run multiple instances, all you need for a new

ZooKeeper instance, even if it's on the same machine for testing purposes, is a new configuration file. To complete the example you'll create two more configuration files.

The `<ZOOKEEPER_HOME>/conf/zoo2.cfg` file should have the content:

```
tickTime=2000
dataDir=c:/sw/zookeeperdata/2
clientPort=2182
initLimit=5
syncLimit=2
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

You'll also need to create `<ZOOKEEPER_HOME>/conf/zoo3.cfg`:

```
tickTime=2000
dataDir=c:/sw/zookeeperdata/3
clientPort=2183
initLimit=5
syncLimit=2
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

Finally, create your `myid` files in each of the `dataDir` directories so that each server knows which instance it is. The id in the `myid` file on each machine must match the "server.X" definition. So, the ZooKeeper instance (or machine) named "server.1" in the above example, must have a `myid` file containing the value "1". The `myid` file can be any integer between 1 and 255, and must match the server IDs assigned in the `zoo.cfg` file.

To start the servers, you can simply explicitly reference the configuration files:

```
cd <ZOOKEEPER_HOME>
bin/zkServer.sh start zoo.cfg
bin/zkServer.sh start zoo2.cfg
bin/zkServer.sh start zoo3.cfg
```

Once these servers are running, you can reference them from Solr just as you did before:

```
java -DnumShards=2 -Dbootstrap_confdir=./solr/collection1/conf \
     -Dcollection.configName=myconf
-DzkHost=localhost:2181,localhost:2182,localhost:2183 -jar start.jar
```

For more information on getting the most power from your ZooKeeper installation, check out the ZooKeeper Administrator's Guide.

## Using ZooKeeper to Manage Configuration Files

With SolrCloud your configuration files (particularly `solrconfig.xml` and `schema.xml`) are kept in ZooKeeper. These files are uploaded when you first start Solr in SolrCloud mode.

### Startup Bootstrap Parameters

There are two different ways you can use system properties to upload your initial configuration files to ZooKeeper the first time you start Solr. Remember that these are meant to be used only on first startup or when overwriting configuration files. Every time you start Solr with these system properties, any current configuration files in ZooKeeper may be overwritten when `conf.set` names match.

The first way is to look at `solr.xml` and upload the `conf` for each core found. The `config set` name will be the collection name for that core, and collections will use the `config set` that has a matching name. One parameter is used with this approach, `bootstrap_conf`. If you pass `-Dbootstrap_conf=true` on startup, each core you have configured will have its configuration files automatically uploaded and linked to the collection containing the core.

An alternate approach is to upload the given directory as a `config set` with the given name. No linking of collection to `config set` is done. However, if only one `conf.set` exists, a collection will autolink to it. Two parameters are used with this approach:

| Parameter | Default value | Description |
|---|---|---|
| `bootstrap_confdir` | No default | If you pass `-bootstrap_confdir=<directory>` on startup, that specific directory of configuration files will be uploaded to ZooKeeper with a `conf.set` name defined by the system property below, `collection.configName`. |
| `collection.configName` | Defaults to `configuration1` | Determines the name of the `conf.set` pointed to by `bootstrap_confdir`. |

Using the ZooKeeper Command Line Interface (zkCLI), you can download and re-upload these configuration files.

> ⚠ It's important to keep these files under version control.

### Managing Your SolrCloud Configuration Files

To update or change your SolrCloud configuration files:

1. Download the latest configuration files from ZooKeeper, using the source control checkout process.
2. Make your changes.
3. Commit your changed file to source control.
4. Push the changes back to ZooKeeper.
5. Reload the collection so that the changes will be in effect.

There are some scripts available with the ZooKeeper Command Line Utility to help manage changes to configuration files, discussed in the section on Command Line Utilities.

> ⚠ By default, `solr.xml` is not one of the Solr configuration files managed by ZooKeeper. If you would like to keep your `solr.xml` in ZooKeeper, starting with Solr 4.5 you can push it to ZooKeeper with the `zkcli.sh` utility (using the `putfile` command). See the section Command Line Utilities for more information.

## Collections API

The Collections API is used to enable you to create, remove, or reload collections, but in the context of SolrCloud you can also use it to create collections with a specific number of shards and replicas.

### API Entry Points

The base URL for all API calls below is `http://<hostname>:<port>/solr`.

`/admin/collections?action=CREATE`: create a collection
`/admin/collections?action=RELOAD`: reload a collection
`/admin/collections?action=SPLITSHARD`: split a shard into two new shards
`/admin/collections?action=CREATESHARD`: create a new shard
`/admin/collections?action=DELETESHARD`: delete an inactive shard
`/admin/collections?action=CREATEALIAS`: create or modify an alias for a collection
`/admin/collections?action=DELETEALIAS`: delete an alias for a collection
`/admin/collections?action=DELETE`: delete a collection
`/admin/collections?action=DELETEREPLICA`: delete a replica of a shard
`/admin/collections?action=ADDREPLICA`: add a replica of a shard
`/admin/collections?action=CLUSTERPROP`: Add/edit/delete a cluster-wide property
`/admin/collections?action=MIGRATE:` Migrate documents to another collection
`/admin/collections?action=ADDROLE:` Add a specific role to a node in the cluster
`/admin/collections?action=REMOVEROLE:` Remove an assigned role
`/admin/collections?action=OVERSEERSTATUS:` Get status and statistics of the overseer
`/admin/collections?action=CLUSTERSTATUS:` Get cluster status
`/admin/collections?action=REQUESTSTATUS`: Get the status of a previous asynchronous request
`/admin/collections?action=LIST:` List all collections

### Create a Collection

`/admin/collections?action=CREATE&name=` *name* `&numShards=` *number* `&replicationFactor=` *number* `&maxShardsPerNode=` *number* `&createNodeSet=` *nodelist* `&collection.configName=` *configname*

### Input

**Query Parameters**

| Key | Type | Required | Default | Description |
|---|---|---|---|---|
| name | string | Yes | | The name of the collection to be created. |
| router.name | string | No | compositeId | The router name that will be used. The router defines how documents will be distributed among the shards. The value can be either **implicit**, which uses an internal default hash, or **compositeId**, which allows defining the specific shard to assign documents to. When using the 'implicit' router, the `shards` parameter is required. When using the 'compositeId' router, the `numShards` parameter is required. For more information, see also the section Document Routing. |
| numShards | integer | No | empty | The number of shards to be created as part of the collection. This is a required parameter when using the 'compositeId' router. |

| | | | | |
|---|---|---|---|---|
| shards | string | No | empty | A comma separated list of shard names, e.g., shard-x,shard-y,shard-z . This is a required parameter when using the 'implicit' router. |
| replicationFactor | integer | No | 1 | The number of replicas to be created for each shard. |
| maxShardsPerNode | integer | No | 1 | When creating collections, the shards and/or replicas are spread across all available (i.e., live) nodes, and two replicas of the same shard will never be on the same node. If a node is not live when the CREATE operation is called, it will not get any parts of the new collection, which could lead to too many replicas being created on a single live node. Defining `maxShardsPerNode` sets a limit on the number of replicas CREATE will spread to each node. If the entire collection can not be fit into the live nodes, no collection will be created at all. |
| createNodeSet | string | No | empty | Allows defining the nodes to spread the new collection across. If not provided, the CREATE operation will create shard-replica spread across all live Solr nodes. The format is a comma-separated list of node_names, such as `localhost:8983_solr, localhost:8984_solr, localhost:8985_solr`. |
| collection.configName | string | No | empty | Defines the name of the configurations (which must already be stored in ZooKeeper) to use for this collection. If not provided, Solr will default to the collection name as the configuration name. |
| router.field | string | No | empty | If this field is specified, the router will look at the value of the field in an input document to compute the hash and identify a shard instead of looking at the `uniqueKey` field. If the field specified is null in the document, the document will be rejected. Please note that RealTime Get or retrieval by id would also require the parameter `_route_` (or `shard.keys`) to avoid a distributed search. |
| property.*name*=*value* | string | No | | Set core property *name* to *value*. See core.properties file contents. |
| autoAddReplicas | boolean | No | false | When set to true, enables auto addition of replicas on shared file systems. Settings and overrides: Running Solr on HDFS#AutoAddReplica Settings |
| async | string | No | | Request ID to track this action which will be processed asynchronously |

### Output

#### Output Content

The response will include the status of the request and the new core names. If the status is anything other than "success", an error message will explain why the request failed.

### Examples

#### Input

```
http://localhost:8983/solr/admin/collections?action=CREATE&name=newCollection&numShard
s=2&replicationFactor=1
```

#### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">3764</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3450</int>
      </lst>
      <str name="core">newCollection_shard1_replica1</str>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3597</int>
      </lst>
      <str name="core">newCollection_shard2_replica1</str>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
  </lst>
</response>
```

## Reload a Collection

`/admin/collections?action=RELOAD&name=` *`name`*

The RELOAD action is used when you have changed a configuration in ZooKeeper.

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | The name of the collection to reload. |

### Output

#### Output Content

The response will include the status of the request and the cores that were reloaded. If the status is anything other than "success", an error message will explain why the request failed.

### Examples

#### Input

```
http://localhost:8983/solr/admin/collections?action=RELOAD&name=newCollection
```

#### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1551</int>
  </lst>
  <lst name="success">
    <lst name="10.0.1.6:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">761</int>
      </lst>
    </lst>
    <lst name="10.0.1.4:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1527</int>
      </lst>
    </lst>
  </lst>
</response>
```

#### Split a Shard

/admin/collections?action=SPLITSHARD&collection=*name*&shard=*shardID*

Splitting a shard will take an existing shard and break it into two pieces. The original shard will continue to contain the same data as-is but it will start re-routing requests to the new shards. The new shards will have as many replicas as the original shard. After splitting a shard, you should issue a commit to make the documents visible, and then you can remove the original shard (with the Core API or Solr Admin UI) when ready.

This command allows for seamless splitting and requires no downtime. A shard being split will continue to accept query and indexing requests and will automatically start routing them to the new shards once this operation is complete. This command can only be used for SolrCloud collections created with "numShards" parameter, meaning collections which rely on Solr's hash-based routing mechanism.

The split is performed by dividing the original shard's hash range into two equal partitions and dividing up the documents in the original shard according to the new sub-ranges.

One can also specify an optional 'ranges' parameter to divide the original shard's hash range into arbitrary hash range intervals specified in hexadecimal. For example, if the original hash range is 0-1500 then adding the parameter: ranges=0-1f4,1f5-3e8,3e9-5dc will divide the original shard into three shards with hash range 0-500, 501-1000 and 1001-1500 respectively.

Another optional parameter 'split.key' can be used to split a shard using a route key such that all documents of the specified route key end up in a single dedicated sub-shard. Providing the 'shard' parameter is not required in this case because the route key is enough to figure out the right shard. A route key which spans more than one shard is not supported. For example, suppose split.key=A! hashes to the range 12-15 and belongs to shard 'shard1' with range 0-20 then splitting by this route key would yield three sub-shards with ranges 0-11, 12-15 and 16-20. Note that the sub-shard with the hash range of the route key may also contain documents for other route keys whose hash ranges overlap.

Shard splitting can be a long running process. In order to avoid timeouts, starting Solr 4.8, you can run this as an asynchronous call.

## Input

### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the collection that includes the shard to be split. |
| shard | string | Yes | The name of the shard to be split. |
| ranges | string | No | A comma-separated list of hash ranges in hexadecimal e.g. ranges=0-1f4,1f5-3e8,3e9-5dc |
| split.key | string | No | The key to use for splitting the index |
| property.*name*=*value* | string | No | Set core property *name* to *value*. See core.properties file contents. |
| async | string | No | Request ID to track this action which will be processed asynchronously |

## Output

### Output Content

The output will include the status of the request and the new shard names, which will use the original shard as their basis, adding an underscore and a number. For example, "shard1" will become "shard1_0" and "shard1_1". If the status is anything other than "success", an error message will explain why the request failed.

## Examples

### Input

Split shard1 of the "anotherCollection" collection.

```
http://10.0.1.6:8983/solr/admin/collections?action=SPLITSHARD&collection=anotherCollec
tion&shard=shard1
```

### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">6120</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3673</int>
      </lst>
      <str name="core">anotherCollection_shard1_1_replica1</str>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3681</int>
      </lst>
      <str name="core">anotherCollection_shard1_0_replica1</str>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">6008</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">6007</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">71</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">0</int>
      </lst>
      <str name="core">anotherCollection_shard1_1_replica1</str>
      <str name="status">EMPTY_BUFFER</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">0</int>
      </lst>
      <str name="core">anotherCollection_shard1_0_replica1</str>
      <str name="status">EMPTY_BUFFER</str>
    </lst>
  </lst>
</response>
```

### Create a Shard

Shards can only created with this API for collections that use the 'implicit' router. Use SPLITSHARD for collections using the 'compositeId' router. A new shard with a name can be created for an existing 'implicit' collection.

`/admin/collections?action=CREATESHARD&shard=`*`shardName`*`&collection=`*`name`*

## Input

### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the collection that includes the shard that will be splitted. |
| shard | string | Yes | The name of the shard to be created. |
| createNodeSet | string | No | Allows defining the nodes to spread the new collection across. If not provided, the CREATE operation will create shard-replica spread across all live Solr nodes. The format is a comma-separated list of node_names, such as `localhost:8983_solr`, `localhost:8984_solr`, `localhost:8985_solr`. |
| property.*name*=*value* | string | No | Set core property *name* to *value*. See core.properties file contents. |

## Output

### Output Content

The output will include the status of the request. If the status is anything other than "success", an error message will explain why the request failed.

## Examples

### Input

Create 'shard-z' for the "anImplicitCollection" collection.

```
http://10.0.1.6:8983/solr/admin/collections?action=CREATESHARD&collection=anImplicitCo
llection&shard=shard-z
```

### Output

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">558</int>
   </lst>
</response>
```

### Delete a Shard

Deleting a shard will unload all replicas of the shard and remove them from `clusterstate.json`. It will only remove shards that are inactive, or which have no range given for custom sharding.

`/admin/collections?action=DELETESHARD&shard=`*`shardID`*`&collection=`*`name`*

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the collection that includes the shard to be deleted. |
| shard | string | Yes | The name of the shard to be deleted. |

### Output

#### Output Content

The output will include the status of the request. If the status is anything other than "success", an error message will explain why the request failed.

### Examples

#### Input

Delete 'shard1' of the "anotherCollection" collection.

```
http://10.0.1.6:8983/solr/admin/collections?action=DELETESHARD&collection=anotherColle
ction&shard=shard1
```

#### Output

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">558</int>
   </lst>
   <lst name="success">
      <lst name="10.0.1.4:8983_solr">
         <lst name="responseHeader">
            <int name="status">0</int>
            <int name="QTime">27</int>
         </lst>
      </lst>
   </lst>
</response>
```

## Create or modify an Alias for a Collection

The `CREATEALIAS` action will create a new alias pointing to one or more collections. If an alias by the same name already exists, this action will replace the existing alias, effectively acting like an atomic "MOVE" command.

`/admin/collections?action=CREATEALIAS&name=`*`name`*`&collections=`*`collectionlist`*

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | The alias name to be created. |

|  |  |  |  |
|---|---|---|---|
| collections | string | Yes | The list of collections to be aliased, separated by commas. |

## Output

### Output Content

The output will simply be a responseHeader with details of the time it took to process the request. To confirm the creation of the alias, you can look in the Solr Admin UI, under the Cloud section and find the `aliases.json` file.

## Examples

### Input

Create an alias named "testalias" and link it to the collections named "anotherCollection" and "testCollection".

```
http://10.0.1.6:8983/solr/admin/collections?action=CREATEALIAS&name=testalias&collecti
ons=anotherCollection,testCollection
```

### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">122</int>
  </lst>
</response>
```

#### Delete a Collection Alias

`/admin/collections?action=DELETEALIAS&name=`*`name`*

## Input

### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | The name of the alias to delete. |

## Output

### Output Content

The output will simply be a responseHeader with details of the time it took to process the request. To confirm the removal of the alias, you can look in the Solr Admin UI, under the Cloud section, and find the `aliases.json` file.

## Examples

### Input

Remove the alias named "testalias".

```
http://10.0.1.6:8983/solr/admin/collections?action=DELETEALIAS&name=testalias
```

### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">117</int>
  </lst>
</response>
```

#### Delete a Collection

`/admin/collections?action=DELETE&name=`*`collection`*

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | The name of the collection to delete. |

### Output

#### Output Content

The response will include the status of the request and the cores that were deleted. If the status is anything other than "success", an error message will explain why the request failed.

### Examples

#### Input

Delete the collection named "newCollection".

```
http://10.0.1.6:8983/solr/admin/collections?action=DELETE&name=newCollection
```

#### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">603</int>
  </lst>
  <lst name="success">
    <lst name="10.0.1.6:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">19</int>
      </lst>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
    <lst name="10.0.1.4:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">67</int>
      </lst>
      <str name="saved">/Applications/solr-4.3.0/example/solr/solr.xml</str>
    </lst>
  </lst>
</response>
```

#### Delete a Replica

`/admin/collections?action=DELETEREPLICA&collection=`*`collection`*`&shard=`*`shard`*`&replica=`*`replica`*

Delete a replica from a given collection and shard. If the corresponding core is up and running the core is unloaded and the entry is removed from the clusterstate. If the node/core is down , the entry is taken off the clusterstate and if the core comes up later it is automatically unregistered.

### Input

**Query Parameters**

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the collection. |
| shard | string | Yes | The name of the shard that includes the replica to be removed. |
| replica | string | Yes | The name of the replica to remove. |

### Examples

**Input**

```
http://10.0.1.6:8983/solr/admin/collections?action=DELETEREPLICA&collection=test2&shar
d=shard2&replica=core_node3
```

### Output

**Output Content**

```
<response>
   <lst name="responseHeader"><int name="status">0</int><int
name="QTime">110</int></lst>
</response>
```

#### Add Replica

/admin/collections?action=ADDREPLICA&collection=*collection*&shard=*shard*&node=*solr_node_name*

Add a replica to a shard in a collection. The node name can be specified if the replica is to be created in a specific node

### Input

**Query Parameters**

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the collection. |
| shard | string | No | The name of the shard to which replica is to be added. Either shard or _route_ must be provided |
| _route_ | string | No | If the shard name is not known, just pass the _route_ value and the system would identify the name of the shard |
| node | string | No | The name of the node where the replica should be created |
| instanceDir | string | No | The instanceDir for the core that will be created |
| dataDir | string | No | The directory in which the core should be created |
| property.*name*=*value* | string | No | Set core property *name* to *value*. See core.properties file contents. |
| async | string | No | Request ID to track this action which will be processed asynchronously |

### Examples

**Input**

```
http://10.0.1.6:8983/solr/admin/collections?action=ADDREPLICA&collection=test2&shard=s
hard2&node=192.167.1.2:8983_solr
```

### Output

**Output Content**

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">3764</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3450</int>
      </lst>
      <str name="core">test2_shard2_replica4</str>
      <str name="saved">/Applications/solr-4.8.0/example/solr/solr.xml</str>
    </lst>
</response>
```

#### Cluster Properties

`/admin/collections?action=CLUSTERPROP&name=`*`propertyName`*`&val=`*`propertyValue`*

Add, edit or delete a cluster-wide property.

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| name | string | Yes | The name of the property. The are a set of property names which are allowed. Other names are rejected with an error. As of Solr 4.7, only the property `urlScheme` is supported. |
| val | string | Yes | The value of the property. If the value is empty or null, the property is unset. |

### Output

#### Output Content

The response will include the status of the request and the properties that were updated or removed. If the status is anything other than "0", an error message will explain why the request failed.

### Examples

#### Input

```
http://localhost:8983/solr/admin/collections?action=CLUSTERPROP&name=urlScheme&val=htt
ps://
```

#### Output

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
   </lst>
</response>
```

### Migrate documents to another collection

/admin/collections?action=MIGRATE&collection=*name*&split.key=*key1*!&target.collection=*target_collection*&forward.timeout=60

The MIGRATE command is used to migrate all documents having the given routing key to another collection. The source collection will continue to have the same data as-is but it will start re-routing write requests to the target collection for the number of seconds specified by the forward.timeout parameter. It is the responsibility of the user to switch to the target collection for reads and writes after the ‘migrate’ command completes.

The routing key specified by the ‘split.key’ parameter may span multiple shards on both the source and the target collections. The migration is performed shard-by-shard in a single thread. One or more temporary collections may be created by this command during the ‘migrate’ process but they are cleaned up at the end automatically.

This is a synchronous operation and therefore keeping a large read timeout on the invocation is advised. The request may still timeout due to inherent limitations of the Collection APIs but that doesn’t necessarily mean that the operation has failed. Users should check logs, cluster state, source and target collections before invoking the operation again.

This command works only with collections having the compositeId router. The target collection must not receive any writes during the time the migrate command is running otherwise some writes may be lost.

Please note that the migrate API does not perform any de-duplication on the documents so if the target collection contains documents with the same uniqueKey as the documents being migrated then the target collection will end up with duplicate documents.

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | Yes | The name of the source collection from which documents will be split. |
| target.collection | string | Yes | The name of the target collection to which documents will be migrated. |
| split.key | string | Yes | The routing key prefix. For example, if uniqueKey is a!123, then you would use `split.key=a!`. |
| forward.timeout | int | No | The timeout, in seconds, until which write requests made to the source collection for the given `split.key` will be forwarded to the target shard. The default is 60 seconds. |
| property.*name* =*value* | string | No | Set core property *name* to *value*. See core.properties file contents. |
| async | string | No | Request ID to track this action which will be processed asynchronously |

## Output

### Output Content

The response will include the status of the request.

## Examples

### Input

```
http://localhost:8983/solr/admin/collections?action=MIGRATE&collection=test1&split.key
=a!&target.collection=test2
```

### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">19014</int>
  </lst>
  <lst name="success">
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1</int>
      </lst>
      <str name="core">test2_shard1_0_replica1</str>
      <str name="status">BUFFERING</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">2479</int>
      </lst>
      <str name="core">split_shard1_0_temp_shard1_0_shard1_replica1</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1002</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">21</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1655</int>
      </lst>
      <str name="core">split_shard1_0_temp_shard1_0_shard1_replica2</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">4006</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">17</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1</int>
      </lst>
      <str name="core">test2_shard1_0_replica1</str>
      <str name="status">EMPTY_BUFFER</str>
    </lst>
    <lst name="192.168.43.52:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">31</int>
      </lst>
    </lst>
    <lst name="192.168.43.52:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">31</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1</int>
      </lst>
      <str name="core">test2_shard1_1_replica1</str>
      <str name="status">BUFFERING</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1742</int>
      </lst>
      <str name="core">split_shard1_1_temp_shard1_1_shard1_replica1</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1002</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">15</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1917</int>
      </lst>
      <str name="core">split_shard1_1_temp_shard1_1_shard1_replica2</str>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">5007</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">8</int>
      </lst>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">1</int>
      </lst>
      <str name="core">test2_shard1_1_replica1</str>
      <str name="status">EMPTY_BUFFER</str>
    </lst>
    <lst name="192.168.43.52:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">30</int>
      </lst>
    </lst>
    <lst name="192.168.43.52:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">30</int>
      </lst>
    </lst>
  </lst>
</response>
```

#### Add Role

`/admin/collections?action=ADDROLE&role=`*`roleName`*`&node=`*`nodeName`*

Assign a role to a given node in the cluster.  The only supported role as of 4.7 is 'overseer' . Use this API to dedicate a particular node as Overseer. Invoke it multiple times to add more nodes. This is useful in large clusters where an Overseer is likely to get overloaded . If available, one among the list of nodes which are assigned the 'overseer' role would become the overseer.  The system would  assign the role to any other node if none of the designated nodes are up and running

### Input

**Query Parameters**

| Key | Type | Required | Description |
|---|---|---|---|
| role | string | Yes | The name of the role. The only supported role as of now is *overseer* |
| node | string | Yes | The name of the node . It is possible to assign a role even before that node is started |

### Output

**Output Content**

The response will include the status of the request and the properties that were updated or removed. If the status is anything other than "0", an error message will explain why the request failed.

### Examples

**Input**

```
http://localhost:8983/solr/admin/collections?action=ADDROLE&role=overseer&node=192.167
.1.2:8983_solr
```

**Output**

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
   </lst>
</response>
```

#### Remove Role

`/admin/collections?action=REMOVEROLE&role=`*`roleName`*`&node=`*`nodeName`*

Remove an assigned role. This API is used to undo the roles assigned using ADDROLE operation

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| role | string | Yes | The name of the role. The only supported role as of now is *overseer* |
| node | string | Yes | The name of the node |

### Output

#### Output Content

The response will include the status of the request and the properties that were updated or removed. If the status is anything other than "0", an error message will explain why the request failed.

### Examples

#### Input

```
http://localhost:8983/solr/admin/collections?action=REMOVEROLE&role=overseer&node=192.
167.1.2:8983_solr
```

#### Output

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">0</int>
   </lst>
</response>
```

#### Overseer status and statistics

`/admin/collections?action=OVERSEERSTATUS`

Returns the current status of the overseer, performance statistics of various overseer APIs as well as last 10 failures per operation type.

### Examples

#### Input:

```
http://localhost:8983/solr/admin/collections?action=OVERSEERSTATUS&wt=json
```

```
{
  "responseHeader":{
    "status":0,
    "QTime":33},
  "leader":"127.0.1.1:8983_solr",
  "overseer_queue_size":0,
  "overseer_work_queue_size":0,
  "overseer_collection_queue_size":2,
  "overseer_operations":[
    "createcollection",{
      "requests":2,
      "errors":0,
      "totalTime":1.010137,
      "avgRequestsPerMinute":0.7467088842794136,
      "5minRateRequestsPerMinute":7.525069023276674,
      "15minRateRequestsPerMinute":10.271274280947182,
      "avgTimePerRequest":0.5050685,
      "medianRequestTime":0.5050685,
      "75thPctlRequestTime":0.519016,
      "95thPctlRequestTime":0.519016,
      "99thPctlRequestTime":0.519016,
      "999thPctlRequestTime":0.519016},
    "removeshard",{
      "requests":1,
      "errors":0,
      "totalTime":0.26784,
      "avgRequestsPerMinute":0.4639267176178192,
      "5minRateRequestsPerMinute":8.179027994326175,
      "15minRateRequestsPerMinute":10.560587086130052,
      "avgTimePerRequest":0.26784,
      "medianRequestTime":0.26784,
      "75thPctlRequestTime":0.26784,
      "95thPctlRequestTime":0.26784,
      "99thPctlRequestTime":0.26784,
      "999thPctlRequestTime":0.26784},
    "updateshardstate",{
      "requests":1,
      "errors":0,
      "totalTime":0.609256,
      "avgRequestsPerMinute":0.43725644039684236,
      "5minRateRequestsPerMinute":8.043840552427673,
      "15minRateRequestsPerMinute":10.502079828515368,
      "avgTimePerRequest":0.609256,
      "medianRequestTime":0.609256,
      "75thPctlRequestTime":0.609256,
      "95thPctlRequestTime":0.609256,
      "99thPctlRequestTime":0.609256,
      "999thPctlRequestTime":0.609256},
    "state",{
      "requests":29,
      "errors":0,
      "totalTime":25.777765,
      "avgRequestsPerMinute":8.911471494053579,
      "5minRateRequestsPerMinute":16.77961791015292,
      "15minRateRequestsPerMinute":21.299616774565774,
      "avgTimePerRequest":0.888888448275862,
      "medianRequestTime":0.646322,
      "75thPctlRequestTime":0.7662585,
      "95thPctlRequestTime":4.9277995,
      "99thPctlRequestTime":6.687749,
      "999thPctlRequestTime":6.687749},
    "createshard",{
      "requests":2,
      "errors":0,
      "totalTime":0.328155,
      "avgRequestsPerMinute":0.8384528317300947,
      "5minRateRequestsPerMinute":15.560264184036232,
      "15minRateRequestsPerMinute":20.772071869612244,
      "avgTimePerRequest":0.1640775,
      "medianRequestTime":0.1640775,
      "75thPctlRequestTime":0.198494,
      "95thPctlRequestTime":0.198494,
      "99thPctlRequestTime":0.198494,
      "999thPctlRequestTime":0.198494},
    "leader",{
      "requests":15,
      "errors":0,
      "totalTime":1.850757,
      "avgRequestsPerMinute":4.664791390089222,
      "5minRateRequestsPerMinute":15.267394345445812,
      "15minRateRequestsPerMinute":20.61365640511346,
      "avgTimePerRequest":0.1233838,
      "medianRequestTime":0.095369,
      "75thPctlRequestTime":0.190858,
      "95thPctlRequestTime":0.245846,
      "99thPctlRequestTime":0.245846,
      "999thPctlRequestTime":0.245846},
    "deletecore",{
      "requests":2,
      "errors":0,
      "totalTime":0.1644,
      "avgRequestsPerMinute":0.9277190814105167,
      "5minRateRequestsPerMinute":16.35805598865235,
      "15minRateRequestsPerMinute":21.121174172260105,
      "avgTimePerRequest":0.0822,
      "medianRequestTime":0.0822,
      "75thPctlRequestTime":0.114723,
      "95thPctlRequestTime":0.114723,
      "99thPctlRequestTime":0.114723,
      "999thPctlRequestTime":0.114723}],
  "collection_operations":[
    "overseerstatus",{
      "requests":5,
      "errors":0,
      "totalTime":16.602856,
      "avgRequestsPerMinute":1.8002951096636433,
      "5minRateRequestsPerMinute":7.878245556506509,
      "15minRateRequestsPerMinute":10.39984320341109,
      "avgTimePerRequest":3.3205712000000003,
      "medianRequestTime":3.42046,
      "75thPctlRequestTime":4.0594019999999995,
      "95thPctlRequestTime":4.563145,
      "99thPctlRequestTime":4.563145,
      "999thPctlRequestTime":4.563145},
    "createalias",{
      "requests":1,
      "errors":0,
      "totalTime":101.364917,
      "avgRequestsPerMinute":8.304550290288862,
      "5minRateRequestsPerMinute":12.0,
      "15minRateRequestsPerMinute":12.0,
      "avgTimePerRequest":101.364917,
      "medianRequestTime":101.364917,
      "75thPctlRequestTime":101.364917,
      "95thPctlRequestTime":101.364917,
      "99thPctlRequestTime":101.364917,
      "999thPctlRequestTime":101.364917},
    "splitshard",{
      "requests":1,
      "errors":1,
      "recent_failures":[{
          "request":{
            "operation":"splitshard",
            "shard":"shard2",
            "collection":"example1"},
          "response":[
            "Operation splitshard caused
exception:","org.apache.solr.common.SolrException:org.apache.solr.common.SolrException
: No shard with the specified name exists: shard2",
            "exception",{
              "msg":"No shard with the specified name exists: shard2",
              "rspCode":400}]}],
      "totalTime":5905.432835,
      "avgRequestsPerMinute":0.8198143044809885,
      "5minRateRequestsPerMinute":8.043840552427673,
      "15minRateRequestsPerMinute":10.502079828515368,
      "avgTimePerRequest":2952.7164175,
      "medianRequestTime":2952.7164175000003,
      "75thPctlRequestTime":5904.384052,
      "95thPctlRequestTime":5904.384052,
      "99thPctlRequestTime":5904.384052,
      "999thPctlRequestTime":5904.384052},
    "createcollection",{
      "requests":2,
      "errors":0,
      "totalTime":6294.35359,
      "avgRequestsPerMinute":0.7466431055563431,
      "5minRateRequestsPerMinute":7.5271593686145355,
      "15minRateRequestsPerMinute":10.271591296400848,
      "avgTimePerRequest":3147.176795,
      "medianRequestTime":3147.1767950000003,
      "75thPctlRequestTime":3387.162793,
      "95thPctlRequestTime":3387.162793,
      "99thPctlRequestTime":3387.162793,
      "999thPctlRequestTime":3387.162793},
    "deleteshard",{
      "requests":1,
      "errors":0,
      "totalTime":320.071335,
      "avgRequestsPerMinute":0.4637771550349566,
      "5minRateRequestsPerMinute":8.179027994326175,
      "15minRateRequestsPerMinute":10.560587086130052,
      "avgTimePerRequest":320.071335,
      "medianRequestTime":320.071335,
      "75thPctlRequestTime":320.071335,
      "95thPctlRequestTime":320.071335,
      "99thPctlRequestTime":320.071335,
      "999thPctlRequestTime":320.071335}],
  "overseer_queue":[
    "peek_wait100",{
      "totalTime":2775.554755,
      "avgRequestsPerMinute":12.440395120289685,
      "5minRateRequestsPerMinute":18.487470843855192,
      "15minRateRequestsPerMinute":22.052847430688917,
      "avgTimePerRequest":69.388868875,
      "medianRequestTime":101.1499165,
      "75thPctlRequestTime":101.43390225,
      "95thPctlRequestTime":101.9976678,
      "99thPctlRequestTime":102.037032,
      "999thPctlRequestTime":102.037032},
    "peek_wait_forever",{
      "totalTime":63247.861899,
      "avgRequestsPerMinute":11.64420509572364,
      "5minRateRequestsPerMinute":31.572546097788198,
      "15minRateRequestsPerMinute":41.688934561096204,
      "avgTimePerRequest":1664.4174183947368,
      "medianRequestTime":636.5281970000001,
      "75thPctlRequestTime":1629.3317682499999,
      "95thPctlRequestTime":13220.58495709999,
      "99thPctlRequestTime":16293.17735,
      "999thPctlRequestTime":16293.17735},
    "remove",{
      "totalTime":92.528385,
      "avgRequestsPerMinute":15.979782864505227,
      "5minRateRequestsPerMinute":33.37988956147563,
      "15minRateRequestsPerMinute":42.49548598991928,
      "avgTimePerRequest":1.7793920192307693,
      "medianRequestTime":1.769479,
      "75thPctlRequestTime":2.22114175,
      "95thPctlRequestTime":3.148778999999998,
      "99thPctlRequestTime":4.393077,
      "999thPctlRequestTime":4.393077},
    "poll",{
      "totalTime":94.686248,
      "avgRequestsPerMinute":15.97973186166097,
      "5minRateRequestsPerMinute":33.37988956147563,
      "15minRateRequestsPerMinute":42.49548598991928,
      "avgTimePerRequest":1.8208893846153844,
      "medianRequestTime":1.819817,
      "75thPctlRequestTime":2.266558,
      "95thPctlRequestTime":3.2130298999999978,
      "99thPctlRequestTime":4.433906,
      "999thPctlRequestTime":4.433906}],
  "overseer_internal_queue":[
    "peek",{
      "totalTime":0.516668,
      "avgRequestsPerMinute":0.30642572162118586,
      "5minRateRequestsPerMinute":6.696421749240565,
      "15minRateRequestsPerMinute":9.879502985109362,
      "avgTimePerRequest":0.516668,
      "medianRequestTime":0.516668,
      "75thPctlRequestTime":0.516668,
      "95thPctlRequestTime":0.516668,
      "99thPctlRequestTime":0.516668,
      "999thPctlRequestTime":0.516668},
    "offer",{
      "totalTime":51.784521,
      "avgRequestsPerMinute":15.979724576198302,
      "5minRateRequestsPerMinute":33.37988956147563,
      "15minRateRequestsPerMinute":42.49548598991928,
      "avgTimePerRequest":0.9958561730769231,
      "medianRequestTime":0.8628875,
      "75thPctlRequestTime":1.1464622500000001,
      "95thPctlRequestTime":1.6499188,
      "99thPctlRequestTime":6.091519,
      "999thPctlRequestTime":6.091519},
    "remove",{
      "totalTime":143.130248,
      "avgRequestsPerMinute":27.6584163855513,
      "5minRateRequestsPerMinute":64.95243565926378,
      "15minRateRequestsPerMinute":84.18442055101546,
      "avgTimePerRequest":1.5903360888888889,
      "medianRequestTime":1.660893,
      "75thPctlRequestTime":2.35234925,
      "95thPctlRequestTime":3.19950245,
      "99thPctlRequestTime":5.01803,
      "999thPctlRequestTime":5.01803},
    "poll",{
      "totalTime":147.837065,
      "avgRequestsPerMinute":27.65837219382363,
      "5minRateRequestsPerMinute":64.95243565926378,
      "15minRateRequestsPerMinute":84.18442055101546,
      "avgTimePerRequest":1.6426340555555554,
      "medianRequestTime":1.6923249999999999,
      "75thPctlRequestTime":2.40090275,
      "95thPctlRequestTime":3.2569366,
      "99thPctlRequestTime":5.062005,
      "999thPctlRequestTime":5.062005}],
  "collection_queue":[
    "remove_event",{
      "totalTime":37.638197,
      "avgRequestsPerMinute":3.9610733603305124,
      "5minRateRequestsPerMinute":9.122591857306068,
      "15minRateRequestsPerMinute":10.928990808126446,
      "avgTimePerRequest":3.421654272727273,
      "medianRequestTime":3.411283,
      "75thPctlRequestTime":4.212892,
      "95thPctlRequestTime":4.720874,
      "99thPctlRequestTime":4.720874,
      "999thPctlRequestTime":4.720874},
    "peek_wait_forever",{
      "totalTime":183048.91735,
      "avgRequestsPerMinute":3.677073912023291,
      "5minRateRequestsPerMinute":1.5867138429776346,
      "15minRateRequestsPerMinute":0.6561136902644256,
      "avgTimePerRequest":15254.076445833334,
      "medianRequestTime":6745.20675,
      "75thPctlRequestTime":27662.958113499997,
      "95thPctlRequestTime":49871.380589,
      "99thPctlRequestTime":49871.380589,
      "999thPctlRequestTime":49871.380589}]}
```

### Cluster Status

`/admin/collections?action=CLUSTERSTATUS`

Fetch the cluster status including collections, shards, replicas as well as collection aliases and cluster properties.

#### Input

**Query Parameters**

| Key | Type | Required | Description |
|---|---|---|---|
| collection | string | No | The collection name for which information is requested. If omitted, information on all collections in the cluster will be returned. |
| shard | string | No | The shard(s) for which information is requested. Multiple shard names can be specified as a comma separated list. |

#### Output

**Output Content**

The response will include the status of the request and the cluster status.

#### Examples

**Input**

```
http://localhost:8983/solr/admin/collections?action=clusterstatus&wt=json
```

**Output**

```
{
  "responseHeader":{
    "status":0,
    "QTime":333},
  "cluster":{
    "collections":{
      "collection1":{
        "shards":{
          "shard1":{
            "range":"80000000-ffffffff",
            "state":"active",
            "replicas":{
              "core_node1":{
                "state":"active",
                "core":"collection1",
                "node_name":"127.0.1.1:8983_solr",
                "base_url":"http://127.0.1.1:8983/solr",
                "leader":"true"},
              "core_node3":{
                "state":"active",
                "core":"collection1",
                "node_name":"127.0.1.1:8900_solr",
                "base_url":"http://127.0.1.1:8900/solr"}}},
          "shard2":{
            "range":"0-7fffffff",
            "state":"active",
            "replicas":{
              "core_node2":{
                "state":"active",
                "core":"collection1",
                "node_name":"127.0.1.1:7574_solr",
                "base_url":"http://127.0.1.1:7574/solr",
                "leader":"true"},
              "core_node4":{
                "state":"active",
                "core":"collection1",
                "node_name":"127.0.1.1:7500_solr",
                "base_url":"http://127.0.1.1:7500/solr"}}}},
        "maxShardsPerNode":"1",
        "router":{"name":"compositeId"},
        "replicationFactor":"1",
        "autoCreated":"true",
        "aliases":["both_collections"]},
      "collection2":{
        "shards":{
          "shard1":{
            "range":"80000000-d554ffff",
            "state":"active",
            "replicas":{"core_node1":{
                "state":"active",
                "core":"collection2_shard1_replica1",
                "node_name":"127.0.1.1:8983_solr",
                "base_url":"http://127.0.1.1:8983/solr",
                "leader":"true"}}},
          "shard2":{
            "range":"d5550000-2aa9ffff",
            "state":"active",
            "replicas":{"core_node2":{
                "state":"active",
                "core":"collection2_shard2_replica1",
                "node_name":"127.0.1.1:7500_solr",
                "base_url":"http://127.0.1.1:7500/solr",
                "leader":"true"}}},
          "shard3":{
            "range":"2aaa0000-7fffffff",
            "state":"active",
            "replicas":{"core_node3":{
                "state":"active",
                "core":"collection2_shard3_replica1",
                "node_name":"127.0.1.1:8900_solr",
                "base_url":"http://127.0.1.1:8900/solr",
                "leader":"true"}}}},
        "maxShardsPerNode":"1",
        "router":{"name":"compositeId"},
        "replicationFactor":"1",
        "autoAddReplicas":"false",
        "aliases":["both_collections"]}},
    "aliases":{"both_collections":"collection1,collection2"},
    "roles":{"overseer":["127.0.1.1:8983_solr",
        "127.0.1.1:7574_solr"]},
    "live_nodes":["127.0.1.1:7574_solr",
       "127.0.1.1:7500_solr",
       "127.0.1.1:8983_solr",
       "127.0.1.1:8900_solr"]}}
```

#### Request Status

`/admin/collections?action=REQUESTSTATUS&requestid=`*`request-id`*

Request the status of an already submitted Asynchronous Collection API call. This call is also used to clear up the stored statuses (See below).

### Input

#### Query Parameters

| Key | Type | Required | Description |
|---|---|---|---|
| requestid | string | Yes | The user defined request-id for the request. This can be used to track the status of the submitted asynchronous task. -1 is a special request id which is used to cleanup the stored states for all of the already completed/failed tasks. |

### Examples

#### Input: Valid Request Status

```
http://localhost:8983/solr/admin/collections?action=REQUESTSTATUS&requestid=1000
```

#### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <lst name="status">
    <str name="state">completed</str>
    <str name="msg">found 1000 in completed tasks</str>
  </lst>
</response>
```

#### Input: Invalid RequestId

```
http://localhost:8983/solr/admin/collections?action=REQUESTSTATUS&requestid=1004
```

#### Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <lst name="status">
    <str name="state">notfound</str>
    <str name="msg">Did not find taskid [1004] in any tasks queue</str>
  </lst>
</response>
```

**Input: Clearing up all the stored statuses**

```
http://localhost:8983/solr/admin/collections?action=REQUESTSTATUS&requestid=-1
```

### List Collections

`/admin/collections?action=LIST`

Fetch the names of the collections in the cluster.

**Example**

**Input**

```
http://localhost:8983/solr/admin/collections?action=LIST&wt=json
```

**Output**

```
{
  "responseHeader":{
    "status":0,
    "QTime":2011},
  "collections":["collection1",
    "example1",
    "example2"]}
```

### Asynchronous Calls

Since some collection API calls can be long running tasks e.g. Shard Split, you can optionally have the calls run asynchronously. Specifying *async=<request-id>* enables you to make an asynchronous call, the status of which can be, at any point requested using the REQUESTSTATUS call.

As of now, the REQUESTSTATUS does not automatically cleanup the tracking data structures i.e. the status of completed/failed tasks stays stored in ZooKeeper unless cleared manually. Sending a REQUESTSTATUS call with *requestid* of ***-1*** clears the stored statuses.

**Example**

**Input**

```
http://localhost:8983/solr/admin/collections?action=SPLITSHARD&collection=collection1&
shard=shard1&async=1000
```

**Output**

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">99</int>
  </lst>
  <str name="requestid">1000</str>
</response>
```

## Parameter Reference

***Cluster Parameters***

| | | |
|---|---|---|
| `numShards` | Defaults to 1 | The number of shards to hash documents to. There must be one leader per shard and each leader can have N replicas. |

***SolrCloud Instance Parameters***

These are set in `solr.xml`, but by default they are set up to also work with system properties.

| | | |
|---|---|---|
| `host` | Defaults to the first local host address found | If the wrong host address is found automatically, you can override the host address with this parameter. |
| `hostPort` | Defaults to the jetty.port system property | The port that Solr is running on. By default this is found by looking at the `jetty.port` system property. |
| `hostContext` | Defaults to solr | The context path for the Solr web application. |

***SolrCloud Instance ZooKeeper Parameters***

| | | |
|---|---|---|
| `zkRun` | Defaults to `localhost:<solrPort+1001>` | Causes Solr to run an embedded version of ZooKeeper. Set to the address of ZooKeeper on this node; this allows us to know who you are in the list of addresses in the `zkHost` connect string. Use -DzkRun to get the default value. |
| `zkHost` | No default | The host address for ZooKeeper. Usually this is a comma-separated list of addresses to each node in your ZooKeeper ensemble. |
| `zkClientTimeout` | Defaults to 15000 | The time a client is allowed to not talk to ZooKeeper before its session expires. |

`zkRun` and `zkHost` are set up using system properties. `zkClientTimeout` is set up in `solr.xml` by default, but can also be set using a system property.

#### *SolrCloud Core Parameters*

| | | |
|---|---|---|
| `shardId` | Defaults to being automatically assigned based on numShards | Allows you to specify the id used to group cores into shards. |

`shardId` can be configured in `solr.xml` for each core element as an attribute.

Additional cloud related parameters are discussed in Solr Cores and solr.xml.

## Command Line Utilities

Solr's Administration page (found by default at `http://hostname:8983/solr/`), provides a section with menu items for monitoring indexing and performance statistics, information about index distribution and replication, and information on all threads running in the JVM at the time. There is also a section where you can run queries, and an assistance area.

In addition, SolrCloud provides its own administration page (found by default at http://localhost:8983/solr/#/~cloud), as well as a few tools available via ZooKeeper's Command Line Utility (CLI). The CLI lets you upload configuration information to ZooKeeper, in the same two ways that were shown in the examples in Parameter Reference. It also provides a few other commands that let you link collection sets to collections, make ZooKeeper paths or clear them, and download configurations from ZooKeeper to the local filesystem.

### Using The ZooKeeper CLI

ZooKeeper has a utility that lets you pass command line parameters: `zkcli.bat` (for Windows environments) and `zkcli.sh` (for Unix environments).

#### *`zkcli` Parameters*

| Short | Parameter Usage | Meaning |
|---|---|---|
| | `-cmd <arg>` | CLI Command to be executed: `bootstrap`, `upconfig`, `downconfig`, `linkconfig`, `makepath`, `get`, `getfile`, `put`, `putfile`, `list` or `clear`.<br>This parameter is **mandatory** |
| `-z` | `-zkhost <locations>` | ZooKeeper host address.<br>This parameter is **mandatory** for all CLI commands. |
| `-c` | `-collection <name>` | For `linkconfig`: name of the collection. |
| `-d` | `-confdir <path>` | For `upconfig`: a directory of configuration files. |
| `-h` | `-help` | Display help text. |
| `-n` | `-confname <arg>` | For `upconfig`, `linkconfig`: name of the configuration set. |
| `-r` | `-runzk <port>` | Run ZooKeeper internally by passing the Solr run port; only for clusters on one machine. |

| | | |
|---|---|---|
| -s | -solrhome <path> | For `bootstrap` or when using `-runzk`: the **mandatory** solrhome location. |

The short form parameter options may be specified with a single dash (eg: `-c mycollection`).
The long form parameter options may be specified using either a single dash (eg: `-collection mycollection`) or a double dash (eg: `--collection mycollection`)

### ZooKeeper CLI Examples

Below are some examples of using the `zkcli` CLI:

#### *Uploading a Configuration Directory*

```
java -classpath example/solr-webapp/WEB-INF/lib/*
     org.apache.solr.cloud.ZkCLI -cmd upconfig -zkhost 127.0.0.1:9983
     -confdir example/solr/collection1/conf -confname conf1 -solrhome  example/solr
```

#### *Put arbitrary data into a new ZK file*

```
java -classpath example/solr-webapp/WEB-INF/lib/*
     org.apache.solr.cloud.ZkCLI -zkhost 127.0.0.1:9983 -put /data.txt 'some data'
```

#### *Put a local file into a new ZK file*

```
java -classpath example/solr-webapp/WEB-INF/lib/*
     org.apache.solr.cloud.ZkCLI -zkhost 127.0.0.1:9983 -putfile /data.txt
/some/local/file.txt
```

#### *Linking a Collection to a Configuration Set*

```
java -classpath example/solr-webapp/webapp/WEB-INF/lib/*
     org.apache.solr.cloud.ZkCLI -cmd linkconfig -zkhost 127.0.0.1:9983
     -collection collection1 -confname conf1 -solrhome example/solr
```

#### *Bootstrapping All the Configuration Directories in `solr.xml`*

```
java -classpath example/solr-webapp/webapp/WEB-INF/lib/*
     org.apache.solr.cloud.ZkCLI -cmd bootstrap -zkhost 127.0.0.1:9983
     -solrhome example/solr
```

#### *Scripts*

There are scripts in `example/cloud-scripts` that handle the classpath and class name for you if you are using Solr out of the box with Jetty. Commands then become:

```
sh zkcli.sh -cmd linkconfig -zkhost  127.0.0.1:9983
     -collection collection1 -confname conf1 -solrhome example/solr
```

## SolrCloud with Legacy Configuration Files

All of the required configuration is already set up in the sample configurations shipped with Solr. You only need to add the following if you are migrating old configuration files. Do not remove these files and parameters from a new Solr instance if you intend to use Solr in SolrCloud mode.

These properties exist in 3 files: `schema.xml`, `solrconfig.xml`, and `solr.xml`.

1. In `schema.xml`, you must have a `_version_` field defined:

```
<field name="_version_" type="long" indexed="true" stored="true" multiValued="false"/>
```

2. In `solrconfig.xml`, you must have an `UpdateLog` defined. This should be defined in the `updateHandler` section.

```
<updateHandler>
  ...
  <updateLog>
    <str name="dir">${solr.data.dir:}</str>
  </updateLog>
  ...
</updateHandler>
```

3. You must have a replication handler called `/replication` defined:

```
<requestHandler name="/replication" startup="lazy" />
```

There are several parameters available for this handler, discussed in the section Index Replication.

4. You must have a Realtime Get handler called "/get" defined:

```
<requestHandler name="/get">
  <lst name="defaults">
    <str name="omitHeader">true</str>
  </lst>
</requestHandler>
```

The parameters for this handler are discussed in the section RealTime Get.

5. You must have the admin handlers defined:

```
<requestHandler name="/admin/" class="solr.admin.AdminHandlers" />
```

6. And you must leave the admin path in `solr.xml` as the default:

```
<cores adminPath="/admin/cores" />
```

7. The DistributedUpdateProcessor is part of the default update chain and is automatically injected into any of your custom update chains, so you don't actually need to make any changes for this capability. However, should you wish to add it explicitly, you can still add it to the `solrconfig.xml` file as part of an `updateRequestProcessorChain`. For example:

```
<updateRequestProcessorChain name="sample">
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory"/>
  <processor class="my.package.UpdateFactory"/>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

If you do not want the DistributedUpdateProcessFactory auto-injected into your chain (for example, if you want to use SolrCloud functionality, but you want to distribute updates yourself) then specify the `NoOpDistributingUpdateProcessorFactory` update processor factory in your chain:

```
<updateRequestProcessorChain name="sample">
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.NoOpDistributingUpdateProcessorFactory"/>
  <processor class="my.package.MyDistributedUpdateFactory"/>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

In the update process, Solr skips updating processors that have already been run on other nodes.

# Legacy Scaling and Distribution

This section describes how to set up distribution and replication in Solr. It is considered "legacy" behavior, since while it is still supported in Solr, the SolrCloud functionality described in the previous chapter is where the current development is headed. However, if you don't need all that SolrCloud delivers, search distribution and index replication may be sufficient.

This section covers the following topics:

Introduction to Scaling and Distribution: Conceptual information about distribution and replication in Solr.

Distributed Search with Index Sharding: Detailed information about implementing distributed searching in Solr.

Index Replication: Detailed information about replicating your Solr indexes.

Combining Distribution and Replication: Detailed information about replicating shards in a distributed index.

Merging Indexes: Information about combining separate indexes in Solr.

# Introduction to Scaling and Distribution

Both Lucene and Solr were designed to scale to support large implementations with minimal custom coding. This section covers:

- distributing an index across multiple servers
- replicating an index on multiple servers
- merging indexes

If you need full scale distribution of indexes and queries, as well as replication, load balancing and failover, you may want to use SolrCloud. Full details on configuring and using SolrCloud is available in the section SolrCloud.

### 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.

### What Problem Does Replication Solve?

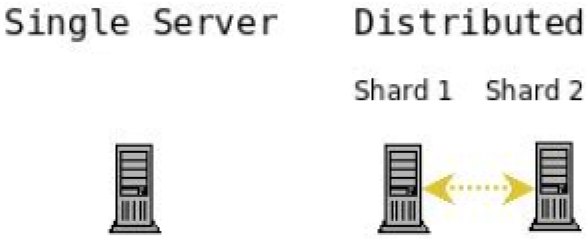
Replicating an index is useful when:

- You have a large search volume which one machine cannot 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 Backing Up).

# Distributed Search with Index Sharding

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. You can find additional information about distributed search on the Solr wiki: http://wiki.apache.org/solr/DistributedSearch.

The figure below compares a single server to a distributed configuration with two shards.



> ⓘ If single queries are currently fast enough and if one simply wants 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.

Update commands may be sent to any server with distributed indexing configured correctly. Document adds and deletes are forwarded to the appropriate server/shard based on a hash of the unique document id. **commit** comman ds and **deleteByQuery** commands are sent to every server in `shards`.

Update reorders (i.e., replica A may see update X then Y, and replica B may see update Y then X). **deleteByQuery** also handles reorders the same way, to ensure replicas are consistent.  All replicas of a shard are consistent, even if the updates arrive in a different order on different replicas.

### Distributing Documents across Shards

It is 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 Indexing and Basic Data Operations.)

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 is easy to know where a document is if you need to update it or delete. In contrast, if you are 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 is 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 is probably best to randomly distribute documents to your shards.

You can directly configure aspects of the concurrency and thread-pooling used within distributed search in Solr. This allows for finer grained control and you can tune it to target your own specific requirements. The default configuration favors throughput over latency.

To configure the standard handler, provide a configuration like this:

```
<requestHandler name="standard" class="solr.SearchHandler" default="true">
    <!-- other params go here -->
     <shardHandlerFactory class="HttpShardHandlerFactory">
        <int name="socketTimeOut">1000</int>
        <int name="connTimeOut">5000</int>
      </shardHandler>
  </requestHandler>
```

The parameters that can be specified are as follows:

| Parameter | Default | Explanation |
|---|---|---|
| `socketTimeout` | 0 (use OS default) | The amount of time in ms that a socket is allowed to wait. |
| `connTimeout` | 0 (use OS default) | The amount of time in ms that is accepted for binding / connecting a socket |
| `maxConnectionsPerHost` | 20 | The maximum number of connections that is made to each individual shard in a distributed search. |
| `corePoolSize` | 0 | The retained lowest limit on the number of threads used in coordinating distributed search. |
| `maximumPoolSize` | Integer.MAX_VALUE | The maximum number of threads used for coordinating distributed search. |
| `maxThreadIdleTime` | 5 seconds | The amount of time to wait for before threads are scaled back in response to a reduction in load. |
| `sizeOfQueue` | -1 | If specified, the thread pool will use a backing queue instead of a direct handoff buffer. High throughput systems will want to configure this to be a direct hand off (with -1). Systems that desire better latency will want to configure a reasonable size of queue to handle variations in requests. |
| `fairnessPolicy` | false | Chooses the JVM specifics dealing with fair policy queuing, if enabled distributed searches will be handled in a First in First out fashion at a cost to throughput. If disabled throughput will be favored over latency. |

### 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://localhost: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 is usually preferred to configure this parameter as a default in the RequestHandler section of `solrconfig.xml`.

> 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.

Where `shards.info=true`, distributed responses will include information about the shard (where each shard represents a logically different index or physical location), such as the following:

```
<lst name="shards.info">
  <lst name="localhost:7777/solr">
    <long name="numFound">1333</long>
    <float name="maxScore">1.0</float>
    <str name="shardAddress">http://localhost:7777/solr</str>
    <long name="time">686</long>
  </lst>
  <lst name="localhost:8888/solr">
    <long name="numFound">342</long>
    <float name="maxScore">1.0</float>
    <str name="shardAddress">http://localhost:8888/solr</str>
    <long name="time">602</long>
  </lst>
</lst>
```

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.

### Limitations to Distributed Search

Distributed searching in Solr has the 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 cannot be distributed.
- The index for distributed searching may become momentarily out of sync if a commit happens between the first and second phase of the distributed search. This might cause a situation where a document that once matched a query and was subsequently changed may no longer match the query but will still be retrieved. This situation is expected to be quite rare, however, and is only possible for a single query request.
- 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.

- Shard information can be returned with each document in a distributed search by including `fl=id,[shard]` in the search request. This returns the shard URL.
- In a distributed search, the data directory from the core descriptor overrides any data directory in `solrconfig.xml`.
- Update commands may be sent to any server with distributed indexing configured correctly. Document adds and deletes are forwarded to the appropriate server/shard based on a hash of the unique document id. **commit** commands and **deleteByQuery** commands are sent to every server in `shards`.

### 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.

For example, a deadlock might occur in the 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 will not 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.

### 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.)

1. Make a copy of the solr example directory:

```
cd solr
cp -r example example7574
```

2. Change the port number:

```
perl -pi -e s/8983/7574/g example7574/etc/jetty.xml \
   example7574/exampledocs/post.sh
```

3. In the first window, start up the server on port 8983:

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

4. In the second window, start up the server on port 7574:

```
cd example7574
java -server -jar start.jar
```

5. In the third window, index some example documents to each server:

```
cd example/exampledocs
./post.sh [a-m]*.xml
cd ../../example7574/exampledocs
./post.sh [n-z]*.xml
```

6. Now do a distributed search across both servers with your browser or `curl`:

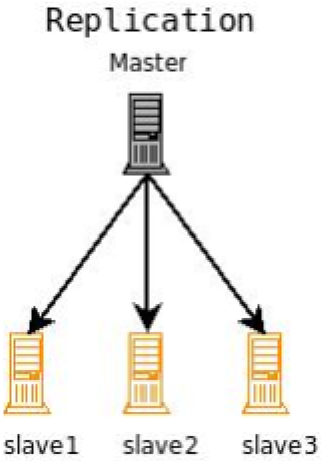
```
curl
'http://localhost:8983/solr/select?shards=localhost:8983/solr,localhost:7574/solr
&indent=true&q=ipod+solr'
```

## Index Replication

> ⚠ The Lucene index format has changed with Solr 4. As a result, once you upgrade, previous versions of Solr will no longer be able to read the rest of your indices. In a master/slave configuration, all searchers/slaves should be upgraded before the master. If the master is updated first, the older searchers will not be able to read the new index format.

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.*

Topics covered in this section:

### Index Replication in Solr

Solr includes a Java implementation of index replication that works over HTTP.

For information on the `ssh/rsync` based replication, see Index Replication using ssh and rsync.

The 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 Java-based replication feature is implemented as a RequestHandler. Configuring replication is therefore similar to any normal RequestHandler.

### 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. |
| 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. |

### Configuring the Replication RequestHandler on a Master Server

Before running a replication, you should set the following parameters on initialization of the handler:

| Name | Description |
|---|---|
| replicateAfter | String specifying action after which replication should occur. Valid values are commit, optimize, or startup. There can be multiple values for this parameter. If you use "startup", you need to have a "commit" and/or "optimize" entry also if you want to trigger replication on future commits or optimizes. |
| backupAfter | String specifying action after which a backup should occur. Valid values are commit, optimize, or startup. There can be multiple values for this parameter. It is not required for replication, it just makes a backup. |
| maxNumberOfBackups | Integer specifying how many backups to keep. This can be used to delete all but the most recent N backups. |
| confFiles | The configuration files to replicate, separated by a comma. |
| commitReserveDuration | If your commits are very frequent and your network is slow, you can tweak this parameter to increase the amount of time taken to download 5Mb from the master to a slave. The default is 10 seconds. |

The example below shows how to configure the Replication RequestHandler on a master server.

```
<requestHandler name="/replication" class="solr.ReplicationHandler" >
    <lst name="master">
        <str name="replicateAfter">optimize</str>
        <str name="backupAfter">optimize</str>
        <str name="confFiles">schema.xml,stopwords.txt,elevate.xml</str>
        <str name="commitReserveDuration">00:00:10</str>
    </lst>
    <int name="maxNumberOfBackups">2</int>
</requestHandler>
```

#### 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 ':'.

### 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://remote_host: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>
```

⚠ 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.

### 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 to reduce the index download time.

Here is 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>
```

### 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.

| 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</str>
<str name="replicateAfter">commit</str>
<str name="replicateAfter">optimize</str>
```

### 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 of 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 (for example, 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 moved to the live index directory and the file's timestamp is same as its counterpart on the master.
- A commit command is issued on the slave by the Slave's ReplicationHandler and the new index is loaded.

#### Replicating Configuration Files

To replicate 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.

#### 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 asks the core to load the fresh index from the new directory.

#### 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. |
| http://*slave_host:port*/solr/replication?command=fetchindex | Forces the specified slave to fetch a copy of the index from its master.<br>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.<br>request parameters:<br>• `numberToKeep:` request parameter can be used with the backup command unless the `maxNumberOfBackups` initialization parameter has been specified on the handler – in which case `maxNumberOfBackups` is always used and attempts to use the `numberToKeep` request parameter will cause an error.<br>• `name` : (optional) Backup name . The snapshot will be created in a directory called snapshot.<name> within the data directory of the core . By default the name is generated using date in `yyyyMMddHHmmssSSS` format. If `location` parameter is passed , that would be used instead of the data directory<br>• `location` : Backup location |

| | |
|---|---|
| http:// *master_host:port* /solr/replication?command=deletebackup | Delete any backup created using the `backup` command .<br>request parameters:<br>• name: The name of the snapshot . A snapshot with the name snapshot.<name> must exist .If not, an error is thrown<br>• location: Location where the snapshot is created |

## Index Replication using ssh and rsync

Solr supports `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.

## 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 (that is, not relying on `crond` to run them) is perfectly fine.

### 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.

#### 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`. It's recommended 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. |
| 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 Solr Wiki.

#### 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:<br>`0 0 * * * <solr.solr.home>/solr/bin/snapcleaner -D 7`<br>Additional cleanup can always be performed on-demand by running snapcleaner manually. |
| snappuller<br>snapinstaller | On the slave Solr servers, snappuller should be run out of cron regularly 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
```

⚠ Modern cron allows this to be shortened to `*/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.

### Commit and Optimization

On a very large index, adding even a few documents and 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%.

While optimizing has many benefits, a rapidly changing index will not retain those benefits for long, and since optimization is an intensive process, it may be better to consider other options, such as lowering the merge factor (discussed in this Guide in the section on Index Configuration

### 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.

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.

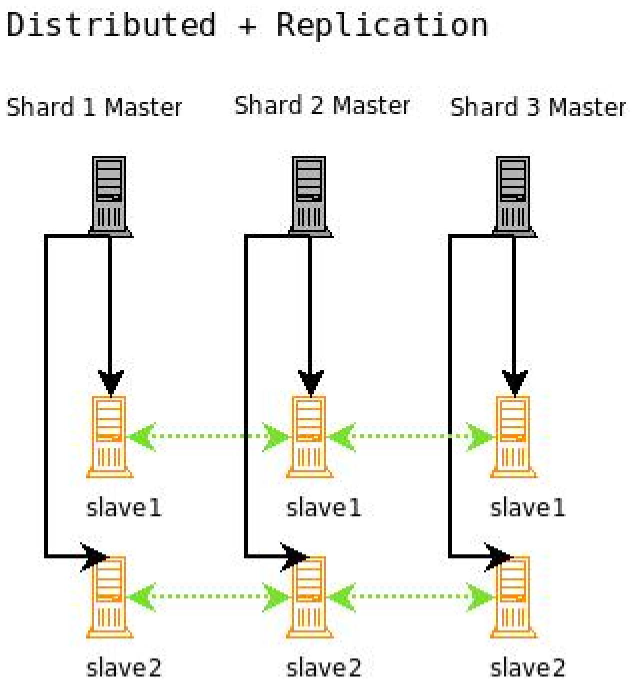
Optimizing on the master allows for a straight-forward optimization operation. No query slaves need to be taken out of service. The optimized collection can be distributed in the background as queries are being normally serviced. The optimization can occur at any time convenient to the application providing collection updates.

## 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.

## 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 either the IndexMergeTool included in `lucene-misc` or the `CoreAdminHandler`.

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.

### Using `IndexMergeTool`

To merge the indexes, do the following:

1. Find the lucene-core and lucene-misc JAR files 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`). These two JAR files should be in `WEB-INF/lib`. They are probably called something like `lucene-core-VERSION.jar` and `lucene-misc-VERSION.jar`.
2. Copy them somewhere easy to find.
3. Make sure that both indexes you want to merge are closed.
4. Issue this command:

   ```
   java -cp /path/to/lucene-core-VERSION.jar:/path/to/lucene-misc-VERSION.jar
    org/apache/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.
5. 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-4.4.0.jar:
   /tmp/lucene-misc-4.4.0.jar org/apache/lucene/misc/IndexMergeTool
    ./newindex
    ./app1/solr/data/index
    ./app2/solr/data/index
   ```

### Using CoreAdmin

This method uses the CoreAdminHandler to execute the `MERGEINDEXES` command with either the `indexDir` or `srcCore` parameters.

The **`indexDir`** parameter is used to define the path to the indexes for the cores that should be merged, and merge them into a 3rd core that must already exist prior to initiation of the merge process. The indexes must exist on the disk of the Solr host, which may make using this in a distributed environment cumbersome. With the `indexDir` parameter, a commit should be called on the cores to be merged (so the IndexWriter will close), and no writes should be allowed on either core until the merge is complete. If writes are allowed, corruption may occur on the merged index. Once complete, a commit should be called on the merged core to make sure the changes are visible to searchers.

The following example shows how to construct the merge command with `indexDir`:

`http://localhost:8983/solr/admin/cores?action=mergeindexes&core=core0&indexDir=/home/solr/core1/data/index&indexDir=/home/solr/core2/data/index`

In this example, `core` is the new core that is created prior to calling the merge process.

The **`srcCore`** parameter is used to call the cores to be merged by name instead of defining the path. The cores do not need to exist on the same disk as the Solr host, and the merged core does not need to exist prior to issuing the command. `srcCore` also protects against corruption during creation of the merged core index, so writes are still possible while the merge occurs. However, `srcCore` can only merge Solr Cores - indexes built directly with Lucene should be merged with either the IndexMergeTool or the `indexDir` parameter.

The following example shows how to construct the merge command with `srcCore`:

```
http://localhost:8983/solr/admin/cores?action=mergeindexes&core=core0&srcCore=core1&srcCore=core2
```

# Client APIs

This section discusses the available client APIs for Solr. It covers the following topics:

Introduction to Client APIs: A conceptual overview of Solr client APIs.

Choosing an Output Format: Information about choosing a response format in Solr.

Using JavaScript: Explains why a client API is not needed for JavaScript responses.

Using Python: Information about Python and JSON responses.

Client API Lineup: A list of all Solr Client APIs, with links.

Using SolrJ: Detailed information about SolrJ, an API for working with Java applications.

Using Solr From Ruby: Detailed information about using Solr with Ruby applications.

MBean Request Handler: Describes the MBean request handler for programmatic access to Solr server statistics and information.

## Introduction to Client APIs

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 also features an EmbeddedSolrServer that offers a Java API without requiring an HTTP connection. For details, see Using SolrJ.

## 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 R esponse Writers.

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.

## Client API Lineup

The Solr Wiki contains a list of client APIs at http://wiki.apache.org/solr/IntegratingSolr.

Here is the list of client APIs, current at this writing (November 2011):

| 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/ |
| SolrNet | .NET | http://code.google.com/p/solrnet/ |
| AJAX Solr | AJAX | http://github.com/evolvingweb/ajax-solr/wiki |

## Using JavaScript

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

## Using Python

Solr includes an output format specifically for Python, but JSON output is a little more robust.

### Simple 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\['name'\]
```

### Python 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\['name'\]
```

## Using SolrJ

SolrJ 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. Then send `SolrRequests` or `SolrQuerys` and get back SolrResponses.

`SolrServer` is abstract, so to connect to a remote Solr instance, you'll actually create an instance of either `HttpSolrServer`, or `CloudSolrServer`.  Both communicate with Solr via HTTP, the different is that `HttpSolrServer` is configured using an explicit Solr URL, while `CloudSolrServer` is configured using the zkHost String for a SolrCl oud cluster

**Single node Solr client**

```
String urlString = "http://localhost:8983/solr";
SolrServer solr = new HttpSolrServer(urlString);
```

**SolrCloud client**

```
String zkHostString = "zkServerA:2181,zkServerB:2181/solr";
SolrServer solr = new CloudSolrServer(zkHostString);
```

Once you have a `SolrServer`, you can use it by calling methods like `query()`, `add()`, and `commit()`.

### Building and Running SolrJ Applications

The SolrJ API is included with Solr, so you do not 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 section require `solr-solrj-4.x.x.jar` to be in the classpath.

At run time, the examples in this section require the libraries found in the 'dist/solrj-lib' directory.

The Ant script bundled with this sections' 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>4.x.0</version>
</dependency>
```

If you are worried about the SolrJ libraries expanding the size of your client application, you can use a code obfuscator like ProGuard to remove APIs that you are not using.

### Setting XMLResponseParser

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());
```

### Performing Queries

Use `query()` to have Solr search for results. You have to pass a `SolrQuery` object that describes the query, and you will 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, submit it with `query()`:

```
QueryResponse response = solr.query(parameters);
```

The client makes a network connection and sends the query. Solr processes the query, and the response is sent and parsed into a `QueryResponse`.

The `QueryResponse` is a collection of documents that satisfy the query parameters. You can retrieve the documents directly with `getResults()` and you can call other methods to find out information about highlighting or facets.

```
SolrDocumentList list = response.getResults();
```

### 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 HttpSolrServer(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);

// Remember to commit your changes!

solr.commit();
```

#### 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, which is the same format SolrJ uses to fetch results.

```
server.setRequestWriter(new BinaryRequestWriter());
```

#### Using the ConcurrentUpdateSolrServer

When implementing java applications that will be bulk loading a lot of documents at once, `ConcurrentUpdateSolrServer` is an alternative to consider instead of using `HttpSolrServer. The`

`ConcurrentUpdateSolrServer` 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 `ConcurrentUpdateSolrServer` for `/update` requests.

### EmbeddedSolrServer

The `EmbeddedSolrServer` class provides an implementation of the `SolrServer` client API talking directly to an micro-instance of Solr running directly in your Java application. This embedded approach is not recommended in most cases and fairly limited in the set of features it supports – in particular it can not be used with SolrCloud or Index Replication. `EmbeddedSolrServer` exists primarily to help facilitate testing.

For information on how to use `EmbeddedSolrServer` please review the SolrJ JUnit tests in the `org.apache.solr.client.solrj.embedded` package of the Solr source release.

### Related Topics

- SolrJ API documentation
- Solr Wiki page on SolrJ
- Indexing and Basic Data Operations

## 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-0.0.8
1 gem installed
Installing ri documentation for solr-ruby-0.0.8...
Installing RDoc documentation for solr-ruby-0.0.8...
```

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.

### 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
```

### Indexing Documents

Indexing is just as simple. You have to get the `Solr::Connection` just as before. Then call the `add()` and `commit()` methods.

```
require 'rubygems'
require 'solr'
solr = Solr::Connection.new('http://localhost:8983/solr')
solr.add(:id => 123, :name => 'Fresh mozzarella cheese')
solr.commit()
```

### More Information

For more information on solr-ruby, read the page at the Solr Wiki:

http://wiki.apache.org/solr/solr-ruby

## MBean Request Handler

The MBean Request Handler offers programmatic access to the information provided on the Plugin/Stats page of the Admin UI. You can access the MBean Request Handler here: http://localhost:8983/solr/admin/mbeans.

The MBean Request Handler accepts the following parameters:

| Parameter | Type | Default | Description |
|---|---|---|---|
| key | multivalued | all | Restricts results by object key. |
| cat | multivalued | all | Restricts results by category name. |
| stats | boolean | false | Specifies whether statistics are returned with results. You can override the `stats` parameter on a per-field basis. |
| wt | multivalued | xml | The output format. This operates the same as the `wt` parameter in a query. |

### Examples

To return information about the CACHE category only:

`http://localhost:8983/solr/admin/mbeans?cat=CACHE`

To return information and statistics about the CACHE category only:

`http://localhost:8983/solr/admin/mbeans?stats=true&cat=CACHE`

To return information for everything, and statistics for everything except the `fieldCache`:

`http://localhost:8983/solr/admin/mbeans?stats=true&f.fieldCache.stats=false`

To return information and statistics for the `fieldCache` only:

`http://localhost:8983/solr/admin/mbeans?key=fieldCache&stats=true`

# Further Assistance

There is a very active user community around Solr and Lucene. The solr-user mailing list, and #solr IRC channel are both great resource for asking questions.

To view the mailing list archives, subscribe to the list, or join the IRC channel, please see https://lucene.apache.org/solr/discussion.html

# Solr Glossary

Where possible, terms are linked to relevant parts of the Solr Reference Guide for more information.

**Jump to a letter:**

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

## A

#### Atomic updates

An approach to updating only one or more fields of a document, instead of reindexing the entire document.

## B

#### Boolean operators

These control the inclusion or exclusion of keywords in a query by using operators such as AND, OR, and NOT.

## C

#### Cluster

In Solr, a cluster is a set of Solr nodes managed as a unit. They may contain many cores, collections, shards, and/or replicas. See also #SolrCloud.

#### Collection

In Solr, one or more documents grouped together in a single logical index. A collection must have a single schema, but can be spread across multiple cores.

In #ZooKeeper, a group of cores managed together as part of a SolrCloud installation.

#### Commit

To make document changes permanent in the index. In the case of added documents, they would be searchable after a *commit*.

#### Core

An individual Solr instance (represents a logical index). Multiple cores can run on a single node. See also #SolrCloud.

#### Core reload

To re-initialize Solr after changes to `schema.xml`, `solrconfig.xml` or other configuration files.

## D

#### Distributed search

Distributed search is one where queries are processed across more than one shard.

#### Document

A group of fields and their values. Documents are the basic unit of data in a collection. Documents are assigned to s hards using standard hashing, or by specifically assigning a shard within the document ID. Documents are versioned after each write operation.

## E

#### Ensemble

A #ZooKeeper term to indicate multiple ZooKeeper instances running simultaneously.

## F

#### Facet

The arrangement of search results into categories based on indexed terms.

#### Field

The content to be indexed/searched along with metadata defining how the content should be processed by Solr.

## I

#### Inverse document frequency (IDF)

A measure of the general importance of a term. It is calculated as the number of total Documents divided by the number of Documents that a particular word occurs in the collection. See http://en.wikipedia.org/wiki/Tf-idf and the Lucene TFIDFSimilarity javadocs for more info on TF-IDF based scoring and Lucene scoring in particular. See also #Term frequency.

#### Inverted index

A way of creating a searchable index that lists every word and the documents that contain those words, similar to an index in the back of a book which lists words and the pages on which they can be found. When performing keyword searches, this method is considered more efficient than the alternative, which would be to create a list of documents paired with every word used in each document. Since users search using terms they expect to be in documents, finding the term before the document saves processing resources and time.

## L

#### Leader

The main node for each shard that routes document adds, updates, or deletes to other replicas in the same shard - this is a transient responsibility assigned to a node via an election, if the current Shard Leader goes down, a new node will be elected to take it's place. See also #SolrCloud.

## M

#### Metadata

Literally, *data about data*. Metadata is information about a document, such as it's title, author, or location.

## N

#### Natural language query

A search that is entered as a user would normally speak or write, as in, "What is aspirin?"

#### Node

A JVM instance running Solr. Also known as a Solr server.

## O

#### Optimistic concurrency

Also known as "optimistic locking", this is an approach that allows for updates to documents currently in the index while retaining locking or version control.

#### Overseer

A single node in SolrCloud that is responsible for processing actions involving the entire cluster. It keeps track of the state of existing nodes and shards, and assigns shards to nodes - this is a transient responsibility assigned to a node via an election, if the current Overseer goes down, a new node will be elected to take it's place. See also #Solr Cloud.

## Q

#### Query parser

A query parser processes the terms entered by a user.

## R

#### Recall

The ability of a search engine to retrieve *all* of the possible matches to a user's query.

#### Relevance

The appropriateness of a document to the search conducted by the user.

#### Replica

A copy of a shard or single logical index, for use in failover or load balancing.

#### Replication

A method of copying a master index from one server to one or more "slave" or "child" servers.

#### RequestHandler

Logic and configuration parameters that tell Solr how to handle incoming "requests", whether the requests are to return search results, to index documents, or to handle other custom situations.

## S

#### SearchComponent

Logic and configuration parameters used by request handlers to process query requests. Examples of search components include faceting, highlighting, and "more like this" functionality.

#### Shard

In SolrCloud, a logical section of a single collection. This may be spread across multiple nodes. See also #SolrCloud

.

#### SolrCloud

Umbrella term for a suite of functionality in Solr which allows managing a cluster of Solr servers for scalability, fault tolerance, and high availability.

#### Solr Schema (schema.xml)

The Apache Solr index schema. The schema defines the fields to be indexed and the type for the field (text, integers, etc.) The schema is stored in schema.xml and is located in the Solr home conf directory.

#### SolrConfig (solrconfig.xml)

The Apache Solr configuration file. Defines indexing options, RequestHandlers, highlighting, spellchecking and various other configurations. The file, solrconfig.xml is located in the Solr home conf directory.

#### Spell Check

The ability to suggest alternative spellings of search terms to a user, as a check against spelling errors causing few or zero results.

#### Stopwords

Generally, words that have little meaning to a user's search but which may have been entered as part of a natural language query. Stopwords are generally very small pronouns, conjunctions and prepositions (such as, "the", "with", or "and")

#### Suggester

Functionality in Solr that provides the ability to suggest possible query terms to users as they type.

#### Synonyms

Synonyms generally are terms which are near to each other in meaning and may substitute for one another. In a search engine implementation, synonyms may be abbreviations as well as words, or terms that are not consistently hyphenated. Examples of synonyms in this context would be "Inc." and "Incorporated" or "iPod" and "i-pod".

## T

#### Term frequency

The number of times a word occurs in a given document. See http://en.wikipedia.org/wiki/Tf-idf and the Lucene TFIDFSimilarity javadocs for more info on TF-IDF based scoring and Lucene scoring in particular.
See also #Inverse document frequency (IDF).

#### Transaction log

An append-only log of write operations maintained by each node. This log is only required with SolrCloud implementations and is created and managed automatically by Solr.

## W

#### Wildcard

A wildcard allows a substitution of one or more letters of a word to account for possible variations in spelling or tenses.

## Z

### ZooKeeper

Also known as Apache ZooKeeper. The system used by SolrCloud to keep track of configuration files and node names for a cluster. A ZooKeeper cluster is used as the central configuration store for the cluster, a coordinator for operations requiring distributed synchronization, and the system of record for cluster topology. See also #SolrCloud.

# Major Changes from Solr 3 to Solr 4

Solr 4 includes some exciting new developments, and also includes many changes from Solr 3.x and earlier.

## Highlights of Solr 4

Solr 4 is a major release of Solr, two years in the making, and includes new features for scalability and high performance for today's data driven, real time search applications. Some of the major improvements include:

### SolrCloud

The primary new feature in Solr 4 goes by the name "SolrCloud", a suite of tools to make scalability built into your project from day one:

- Distributed indexing designed from the ground up for near real-time (NRT) and NoSQL features such as realtime-get, optimistic locking, and durable updates.
- High availability with no single points of failure.
- Apache Zookeeper integration for distributed coordination and cluster metadata and configuration storage.
- Immunity to split-brain issues due to Zookeeper's Paxos distributed consensus protocols.
- Updates sent to any node in the cluster and are automatically forwarded to the correct shard and replicated to multiple nodes for redundancy.
- Queries sent to any node automatically perform a full distributed search across the cluster with load balancing and fail-over.

### NoSQL Features

Users wishing to use Solr as their primary data store will be interested in these features:

- Update durability - A transaction log ensures that even uncommitted documents are never lost.
- Real-time Get - The ability to quickly retrieve the latest version of a document, without the need to commit or open a new searcher
- Versioning and Optimistic Locking - combined with real-time get, this allows read-update-write functionality that ensures no conflicting changes were made concurrently by other clients.
- Atomic updates - the ability to add, remove, change, and increment fields of an existing document without having to send in the complete document again.

### Other Major Features

There's more:

- Pivot Faceting - Multi-level or hierarchical faceting where the top constraints for one field are found for each top constraint of a different field.
- Pseudo-fields - The ability to alias fields, or to add metadata along with returned documents, such as function query values and results of spatial distance calculations.

- A spell checker implementation that can work directly from the main index instead of creating a sidecar index.
- Pseudo-Join functionality - The ability to select a set of documents based on their relationship to a second set of documents.
- Function query enhancements including conditional function queries and relevancy functions.
- New update processors to facilitate modifying documents prior to indexing.
- A brand new web admin interface, including support for SolrCloud.

## Changes to Consider

There are some major changes in Solr 4 to consider before starting to migrate your configurations and indexes. There are many hundreds of changes, so a thorough review of the changes.txt file in your Solr instance will help you plan migration to Solr 4.

### System Changes

- As of Solr 4.8, Java 1.7 is now required to run Solr. Solr versions 4.0 through 4.7 required Java 1.6.

### Index Format

- The Lucene index format has changed. As a result, once you upgrade to Solr 4, previous versions of Solr will no longer be able to read your indices. In a master/slave configuration, all searchers/slaves should be upgraded before the master. If the master is updated first, older searchers will not be able to read the new index format.

### Query Parsers

- The default logic for the `mm` parameter of the Dismax Query Parser has changed. If no `mm` parameter is specified (either in the query or as a default in `solrconfig.xml`, then the effective value of the `q.op` parameter is used to influence the behavior (whether `q.op` is defined in the query, in `solrconfig.xml`, or from the `defaultOperator` option in `schema.xml`). If `q.op` is effectively "AND" then `mm=100%`. If `q.op` is effectively "OR" then `mm=0%`. If you want to force legacy behavior, set a default value for the `mm` parameter in your `solrconfig.xml` file.

### Schema Configuration

- Due to low level changes to support SolrCloud, the `uniqueKey` field can no longer be populated via `<copyField/>` or `<field default=...>` in `schema.xml`. If you want to have Solr automatically generate a uniqueKey field value when adding documents, use an instance of solr.UUIDUpdateProcessorFactory in their update processor chain. See SOLR-2798 for more details.

- Solr is now much more strict about requiring that the `uniqueKey` feature (if used) must refer to a field which is not multiValued. If you upgrade from an earlier version of Solr and see an error that your `uniqueKey` field "can not be configured to be multivalued" please add `multiValued="false"` to the `<field />` declaration for your `uniqueKey` field.

- Changes to the `HTMLCharFilterFactory`:
  - Known offset bugs have been fixed.
  - The "Mark invalid" exceptions are no longer triggered.
  - Newlines are now substituted instead of spaces for block-level elements; this corresponds more closely to on-screen layout, enables sentence segmentation, and doesn't change the offsets.
  - Supplementary characters in tags are now recognized.

- Accepted tag names have been switched from `[:XID_Start:]` and `[:XID_Continue:]` Unicode properties to the more relaxed `[:ID_Start:]` and `[:ID_Continue:]` properties, in order to broaden the range of recognizable input. (The improved security afforded by the `XID_*` properties is irrelevant to what a `CharFilter` does.)
- More cases of <script> tags are now properly stripped.
- CDATA sections are now recognized.
- No space is substituted for inline tags (e.g. <b>, `<i>`, `<span>`). The old version substituted spaces for all tags.
- Broken MS-Word-generated processing instructions `(? ... /)` instead of `<? ... ?>`) are now handled.
- Uppercase character entities "", "©", "<", ">", "®", and "&" are now recognized and handled as if they were lower case.
- Opening tags with unbalanced quotation marks are now properly stripped.
- Literal "<" and ">" characters in opening tags, regardless of whether they appear inside quotation marks, now inhibit recognition (and stripping) of the tags. The only exception to this is for values of event-handler attributes, e.g. "onClick", "onLoad", "onSelect".
- A newline '\n' is substituted instead of a space for stripped HTML markup.
- Nothing is substituted for opening and closing inline tags - they are simply removed. The list of inline tags is (case insensitively): <a>, <abbr>, <acronym>, <b>, <basefont>, <bdo>, <big>, <cite>, <code>, <dfn>, <em>, <font>, <i>, <img>, <input>, <kbd>, <label>, <q>, <s>, <samp>, <select>, <small>, <span>, <strike>, <strong>, <sub>, <sup>, <textarea>, <tt>, <u>, and <var>.
- HTMLStripCharFilterFactory now handles HTMLStripCharFilter's "escapedTags" feature: opening and closing tags with the given names, including any attributes and their values, are left intact in the output.
- The replacement character U+FFFD is now used to replace numeric character entities for unpaired UTF-16 low and high surrogates (in the range [U+D800-U+DFFF]).
- Properly paired numeric character entities for UTF-16 surrogates are now converted to the corresponding code units.
- The generated scanner's parse method has been changed from the default `yylex()` to `nextChar()`.

### Changes to solrconfig.xml

- The <indexDefaults> and <mainIndex> sections of solrconfig.xml have been discontinued and replaced with the <indexConfig> section. There are also better defaults. When migrating, If you don't know what your old settings mean, delete both the <indexDefaults> and <mainIndex> sections. If you have customized them, put them in the <indexConfig> section with the same syntax as before.

- The `PingRequestHandler` no longer looks for a `<healthcheck>` option in the (legacy) `<admin>` section of `solrconfig.xml`. If you want to take advantage of this feature, configure a `healthcheckFile` initialization parameter directly on the `PingRequestHandler`. As part of this change, relative file paths have been fixed to be resolved against the data directory. The sample `solrconfig.xml` has an example of this configuration.

- The update request parameter to choose the Update Request Processor Chain has been renamed from `update.processor` to `update.chain`. The old parameter was deprecated in Solr 3.x, but now has been removed entirely.

- The VelocityResponseWriter is no longer built into the core. Its jar and dependencies now need to be addressed (via <lib> or solr/home lib inclusion). It also needs to be registered in `solrconfig.xml` like this:

```
<queryResponseWriter name="velocity" class="solr.VelocityResponseWriter"/>
```

### Other Changes

- Two of the SolrServer subclasses in SolrJ have been renamed and replaced. `CommonsHttpSolrServer` is now `HttpSolrServer`, and `StreamingUpdateSolrServer` is now `ConcurrentUpdateSolrServer`.

# Errata

## Errata For This Documentation

Any mistakes found in this documentation after its release will be listed on the on-line version of this page:

https://cwiki.apache.org/confluence/display/solr/Errata

## Errata For Past Versions of This Documentation

Any known mistakes in past releases of this documentation will be noted below.