

Apache Solr Reference Guide

Covering Apache Solr 6.2

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.

Apache and the Apache feather logo are trademarks of The Apache Software Foundation. Apache Lucene, Apache Solr and their respective logos are trademarks of the Apache Software Foundation. Please see the Apache Trademark Policy for more information.

Fonts used in the Apache Solr Reference Guide include Raleway, licensed under the SIL Open Font License, 1.1.

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. 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 when running Solr 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, 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. When running the various examples

mentioned through out this tutorial (i.e., `bin/solr -e techproducts`) the `solr.home` will be a sub directory of `example/` created for you automatically.

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.

Solr Start Script Reference: a complete reference of all of the commands and options available with the bin/solr script.

Installing Solr

This section describes how to install Solr. You can install Solr in any system where 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 1.8 or higher. At a command line, check your Java version like this:

```
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

The exact output will vary, but you need to make sure you meet the minimum version requirement. We also recommend choosing a version that is not end-of-life from its vendor. 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. When getting started, all you need to do is extract the Solr distribution archive to a directory of your choosing. When you're ready to setup Solr for a production environment, please refer to the instructions provided on the Taking Solr to Production page. To keep things simple for now, extract the Solr distribution archive to your local home directory, for instance on Linux, do:

```
$ cd ~/
$ tar zxf solr-x.y.z.tgz
```

Once extracted, you are now ready to run Solr using the instructions provided in the Running Solr section.

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 start
```

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

```
bin\solr.cmd start
```

This will start Solr in the background, listening on port 8983.

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

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 Foreground

Since Solr is a server, it is more common to run it in the background, especially on Unix/Linux. However, to start Solr in the foreground, simply do:

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

If you are running Windows, you can run:

```
bin\solr.cmd start -f
```

Start Solr with a Different Port

To change the port Solr listens on, you can use the `-p` parameter when starting, 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 -p 8983
```

The stop command requires you to specify the port Solr is listening on or you can use the `-all` parameter to stop all running Solr instances.

Start Solr with a Specific Bundled Example

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 "techproducts" example, you would do:

```
$ bin/solr -e techproducts
```

Currently, the available examples you can run are: techproducts, dih, schemaless, and cloud. See the section Running with Example Configurations for details on each example.

> **Getting Started with SolrCloud**
> Running the `cloud` example starts Solr in SolrCloud mode. For more information on starting Solr in cloud mode, see the section Getting Started with SolrCloud.

Check if Solr is Running

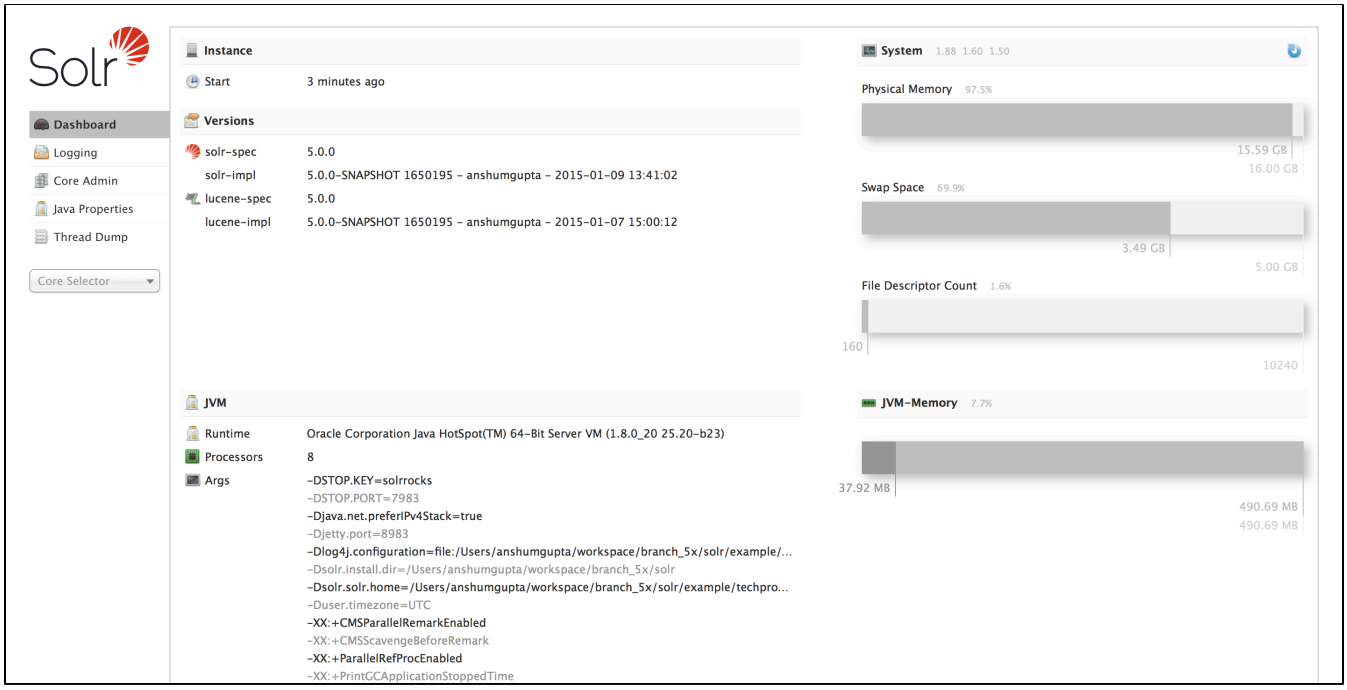
If you're not sure if Solr is running locally, you can use the status command:

```
$ bin/solr status
```

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

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

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



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.

Create a Core

If you did not start Solr with an example configuration, you would need to create a core in order to be able to index and search. You can do so by running:

```
$ bin/solr create -c <name>
```

This will create a core that uses a data-driven schema which tries to guess the correct field type when you add documents to the index.

To see all available options for creating a new core, execute:

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

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 much.

You may want to add a few sample documents before trying to index your own content. The Solr installation comes with different types of example documents located under the sub-directories of the `example/` directory of your installation.

In the `bin/` directory is the post script, a command line tool which can be used to index different types of

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 `bin/post`, use the `-help` option. Windows users, see the section for Post Tool on Windows.

`bin/post` can post various types of content to Solr, including files in Solr's native XML and JSON formats, CSV files, a directory tree of rich documents, or even a simple short web crawl. See the examples at the end of `bin/post -help` for various commands to easily get started posting your content into Solr.

Go ahead and add all the documents in some example XML files:

```
$ bin/post -c gettingstarted example/exampledocs/*.xml
SimplePostTool version 5.0.0
Posting files to [base] url http://localhost:8983/solr/gettingstarted/update...
Entering auto mode. File endings considered are
xml,json,csv,pdf,doc,docx,ppt,pptx,xls,xlsx,odt,odp,ods,ott,otp,ots,rtf,htm,html,txt
,log
POSTing file gb18030-example.xml (application/xml) to [base]
POSTing file hd.xml (application/xml) to [base]
POSTing file ipod_other.xml (application/xml) to [base]
POSTing file ipod_video.xml (application/xml) to [base]
POSTing file manufacturers.xml (application/xml) to [base]
POSTing file mem.xml (application/xml) to [base]
POSTing file money.xml (application/xml) to [base]
POSTing file monitor.xml (application/xml) to [base]
POSTing file monitor2.xml (application/xml) to [base]
POSTing file mp500.xml (application/xml) to [base]
POSTing file sd500.xml (application/xml) to [base]
POSTing file solr.xml (application/xml) to [base]
POSTing file utf8-example.xml (application/xml) to [base]
POSTing file vidcard.xml (application/xml) to [base]
14 files indexed.
COMMITting Solr index changes to http://localhost:8983/solr/gettingstarted/update...
Time spent: 0:00:00.153
```

That's it! Solr has indexed the documents contained in those 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":

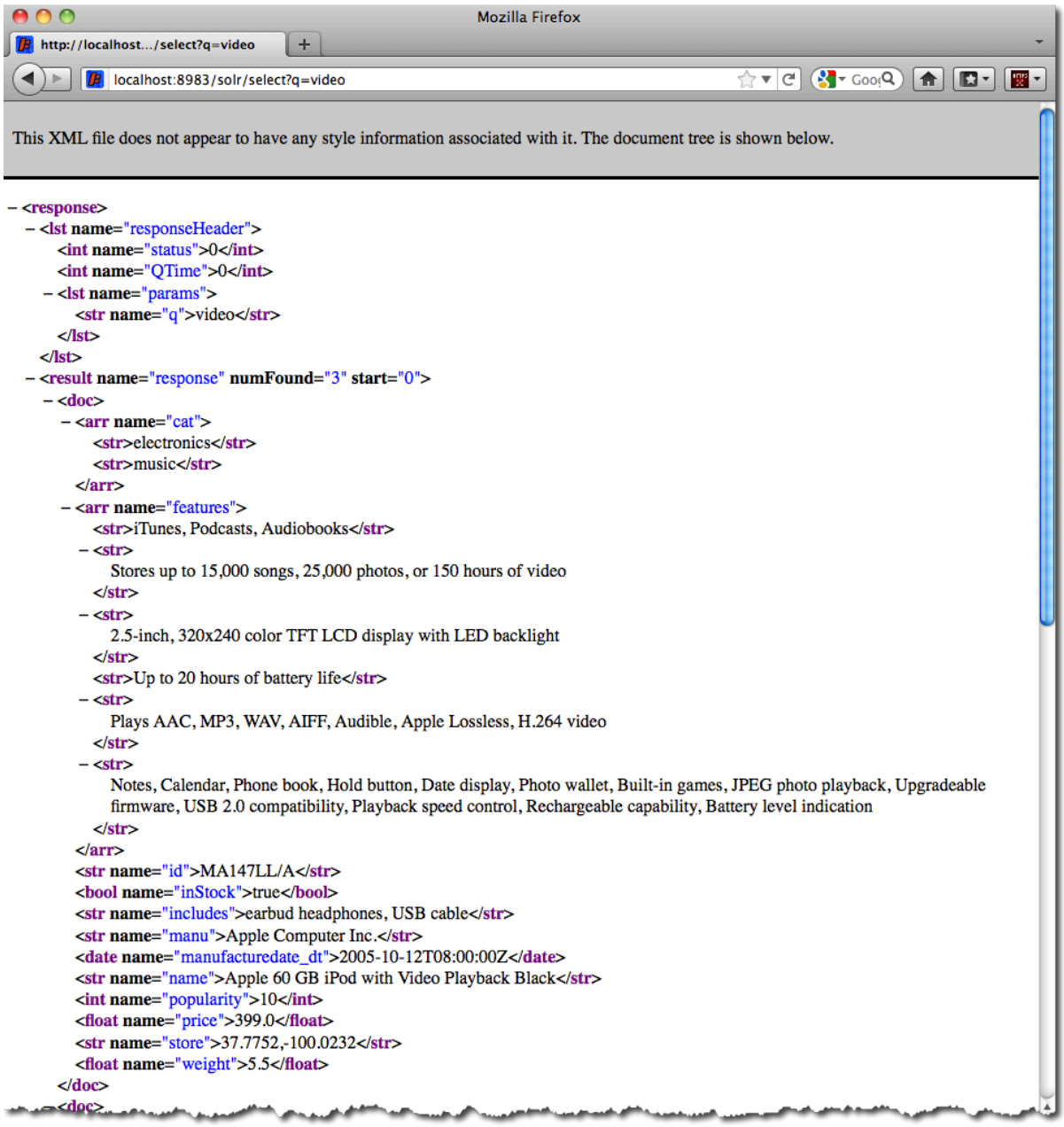
`http://localhost:8983/solr/gettingstarted/select?q=video`

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 `responseHeade`

`r` and a result named response. Inside result, you can see the three docs that represent the search results.



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/gettingstarted/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/gettingstarted/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/gettingstarted/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/gettingstarted/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>
```

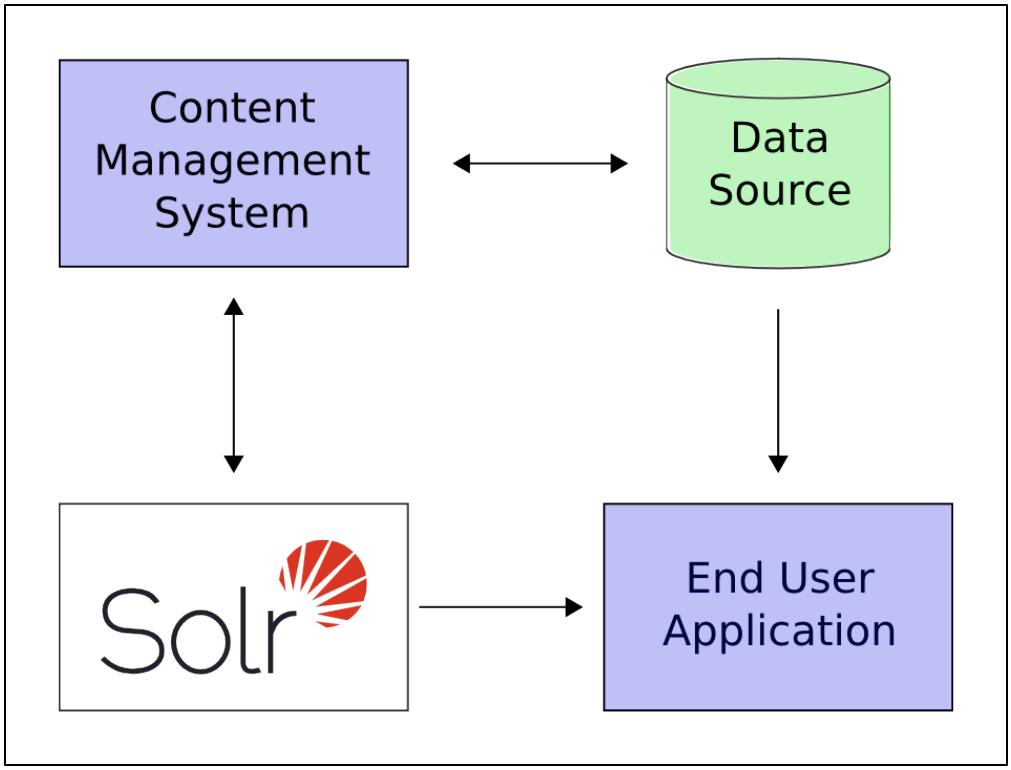
The facet information shows how many of the query results have each possible value of the `cat` field. You could easily use this information to provide users with a quick way to narrow their query results. You can filter results by adding one or more filter queries to the Solr request. This request constrains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

```
http://localhost:8983/solr/gettingstarted/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 example of how Solr might be integrated into an application:



In the scenario above, Solr runs along side other server applications. For example, an online store application would provide a user interface, a shopping cart, and a way to make purchases for end users; while an inventory management application would allow store employees to edit product information. The product metadata would be kept in some kind of database, as well as in Solr.

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.
3. Feed Solr documents 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. Sea

rching 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 much data, or so many queries, that a single Solr server is unable to handle your entire workload. In this case, you can scale up the capabilities of your application using SolrCloud to better distribute the data, and the processing of requests, across many servers. Multiple options can be mixed and matched depending on the type of scalability you need.

For example: "Sharding" is a scaling technique in which a collection is split into multiple logical pieces called "shards" in order to scale up the number of documents in a collection beyond what could physically fit on a single server. Incoming queries are distributed to every shard in the collection, which respond with merged results. Another technique available is to increase the "Replication Factor" of your collection, which allows you to add servers with additional copies of your collection to handle higher concurrent query load by spreading the requests around to multiple machines. Sharding and Replication are not mutually exclusive, and together make Solr an extremely powerful and scalable platform.

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 layout of the home directory will look a little different when you are running Solr in standalone mode vs when you are running in SolrCloud mode.

The crucial parts of the Solr home directory are shown in these examples:

Standalone Mode

```
<solr-home-directory>/
   solr.xml
   core_name1/
      core.properties
      conf/
         solrconfig.xml
         managed-schema
      data/
   core_name2/
      core.properties
      conf/
         solrconfig.xml
         managed-schema
      data/
```

SolrCloud Mode

```
<solr-home-directory>/
   solr.xml
   core_name1/
      core.properties
      data/
   core_name2/
      core.properties
      data/
```

You may see other files, but the main ones you need to know are:

- `solr.xml` specifies configuration options for your Solr server instance. For more information on `solr.xml` see Solr Cores and solr.xml.
- Per Solr Core:
  - `core.properties` defines specific properties for each core such as its name, the collection the core belongs to, the location of the schema, and other parameters. For more details on `core.properties`, see the section Defining core.properties.
  - `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 Configuring solrconfig.xml.
  - `managed-schema` (or `schema.xml` instead) describes the documents you will ask Solr to index. The Schema 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 Solr Schemas, see Documents, Fields, and Schema Design and the Schema API.
  - `data/` The directory containing the low level index files.

Note that the SolrCloud example does not include a `conf` directory for each Solr Core (so there is no `solrconfig.xml` or Schema file). This is because the configuration files usually found in the `conf` directory are stored in ZooKeeper so they can be propagated across the cluster.

If you are using SolrCloud with the embedded ZooKeeper instance, you may also see `zoo.cfg` and `zoo.data` which are ZooKeeper configuration and data files. However, if you are running your own ZooKeeper ensemble, you would supply your own ZooKeeper configuration file when you start it and the copies in Solr would be unused. For more information about ZooKeeper and SolrCloud, see the section SolrCloud.

Solr Start Script Reference

Solr includes a script known as "`bin/solr`" that allows you to start and stop Solr, create and delete collections or cores, and check the status of Solr and configured shards. You can find the script in the `bin/` directory of your Solr installation. The `bin/solr` script makes Solr easier to work with by providing simple commands and options to quickly accomplish common goals.

In this section, the headings below correspond to available commands. For each command, the available options are described with examples.

More examples of bin/solr in use are available throughout the Solr Reference Guide, but particularly in the sections Running Solr and Getting Started with SolrCloud.

- Starting and Stopping
  - Start and Restart
  - Stop

- Informational
  - Version
  - Status
  - Healthcheck
- Collections and Cores
  - Create
  - Delete
- ZooKeeper Operations
  - Uploading a Configuration Set
  - Downloading a Configuration Set
  - Copy between local files and Zookeeper znodes
  - Remove a znode from Zookeeper
  - Move one Zookeeper znode to another (rename)
  - List a Zookeeper znode's children

Starting and Stopping

Start and Restart

The start command starts Solr. The restart command allows you to restart Solr while it is already running or if it has been stopped already.

The start and restart commands have several options to allow you to run in SolrCloud mode, use an example configuration set, start with a hostname or port that is not the default and point to a local ZooKeeper ensemble.

```
bin/solr start [options]
```

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

```
bin/solr restart [options]
```

```
bin/solr restart -help
```

When using the restart command, you must pass all of the parameters you initially passed when you started Solr. Behind the scenes, a stop request is initiated, so Solr will be stopped before being started again. If no nodes are already running, restart will skip the step to stop and proceed to starting Solr.

Available Parameters

The bin/solr script provides many options to allow you to customize the server in common ways, such as changing the listening port. However, most of the defaults are adequate for most Solr installations, especially when just getting started.

Parameter	Description	Example
-a "<string>"	Start Solr with additional JVM parameters, such as those starting with -X. If you are passing JVM parameters that begin with "-D", you can omit the -a option.	`bin/solr start -a "-Xdebug -Xrunjdwp:transport=dt_socket, server=y,suspend=n,address=1044"`

-cloud	Start Solr in SolrCloud mode, which will also launch the embedded ZooKeeper instance included with Solr. This option can be shortened to simply `-c`. If you are already running a ZooKeeper ensemble that you want to use instead of the embedded (single-node) ZooKeeper, you should also pass the -z parameter. For more details, see the section SolrCloud Mode below.	`bin/solr start -c`
-d <dir>	Define a server directory, defaults to `server` (as in, `$SOLR_HOME/server`). It is uncommon to override this option. When running multiple instances of Solr on the same host, it is more common to use the same server directory for each instance and use a unique Solr home directory using the -s option.	`bin/solr start -d newServerDir`
-e <name>	Start Solr with an example configuration. These examples are provided to help you get started faster with Solr generally, or just try a specific feature. The available options are: • cloud • techproducts • dih • schemaless See the section Running with Example Configurations below for more details on the example configurations.	`bin/solr start -e schemaless`
-f	Start Solr in the foreground; you cannot use this option when running examples with the -e option.	`bin/solr start -f`
-h <hostname>	Start Solr with the defined hostname. If this is not specified, 'localhost' will be assumed.	`bin/solr start -h search.mysolr.com`
-m <memory>	Start Solr with the defined value as the min (-Xms) and max (-Xmx) heap size for the JVM.	`bin/solr start -m 1g`

-noprompt	Start Solr and suppress any prompts that may be seen with another option. This would have the side effect of accepting all defaults implicitly. For example, when using the "cloud" example, an interactive session guides you through several options for your SolrCloud cluster. If you want to accept all of the defaults, you can simply add the -noprompt option to your request.	`bin/solr start -e cloud -noprompt`
-p <port>	Start Solr on the defined port. If this is not specified, '8983' will be used.	`bin/solr start -p 8655`
-s <dir>	Sets the solr.solr.home system property; Solr will create core directories under this directory. This allows you to run multiple Solr instances on the same host while reusing the same server directory set using the -d parameter. If set, the specified directory should contain a solr.xml file, unless solr.xml exists in ZooKeeper. The default value is `server/solr`. This parameter is ignored when running examples (-e), as the solr.solr.home depends on which example is run.	`bin/solr start -s newHome`
-V	Start Solr with verbose messages from the start script.	`bin/solr start -V`
-z <zkHost>	Start Solr with the defined ZooKeeper connection string. This option is only used with the -c option, to start Solr in SolrCloud mode. If this option is not provided, Solr will start the embedded ZooKeeper instance and use that instance for SolrCloud operations.	`bin/solr start -c -z server1:2181,server2:2181`

To emphasize how the default settings work take a moment to understand that the following commands are equivalent:

```
bin/solr start
```

```
bin/solr start -h localhost -p 8983 -d server -s solr -m 512m
```

It is not necessary to define all of the options when starting if the defaults are fine for your needs.

Setting Java System Properties

The bin/solr script will pass any additional parameters that begin with -D to the JVM, which allows you to set arbitrary Java system properties. For example, to set the auto soft-commit frequency to 3 seconds, you can do:

```
bin/solr start -Dsolr.autoSoftCommit.maxTime=3000
```

SolrCloud Mode

The -c and -cloud options are equivalent:

```
bin/solr start -c

bin/solr start -cloud
```

If you specify a ZooKeeper connection string, such as `-z 192.168.1.4:2181`, then Solr will connect to ZooKeeper and join the cluster. If you do not specify the -z option when starting Solr in cloud mode, then Solr will launch an embedded ZooKeeper server listening on the Solr port + 1000, i.e., if Solr is running on port 8983, then the embedded ZooKeeper will be listening on port 9983.

IMPORTANT: If your ZooKeeper connection string uses a chroot, such as `localhost:2181/solr`, then you need to bootstrap the /solr znode before launching SolrCloud using the bin/solr script. To do this, you need to use the `zkcli.sh` script shipped with Solr, such as:

```
server/scripts/cloud-scripts/zkcli.sh -zkhost localhost:2181/solr -cmd bootstrap
-solrhome server/solr
```

When starting in SolrCloud mode, the interactive script session will prompt you to choose a configset to use.

For more information about starting Solr in SolrCloud mode, see also the section Getting Started with SolrCloud.

Running with Example Configurations

```
bin/solr start -e <name>
```

The example configurations allow you to get started quickly with a configuration that mirrors what you hope to accomplish with Solr.

Each example launches Solr in with a managed schema, which allows use of the Schema API to make schema edits, but does not allow manual editing of a Schema file If you would prefer to manually modify a `schema.xml` file directly, you can change this default as described in the section Schema Factory Definition in SolrConfig.

Unless otherwise noted in the descriptions below, the examples do not enable SolrCloud nor schemaless mode.

The following examples are provided:

- **cloud**: This example starts a 1-4 node SolrCloud cluster on a single machine. When chosen, an interactive session will start to guide you through options to select the initial configset to use, the number of nodes for your example cluster, the ports to use, and name of the collection to be created. When using this example, you can choose from any of the available configsets found in `$SOLR_HOME/server/solr/configsets`.
- **techproducts**: This example starts Solr in standalone mode with a schema designed for the sample documents included in the `$SOLR_HOME/example/exampledocs` directory. The configset used can be found in `$SOLR_HOME/server/solr/configsets/sample_techproducts_configs`.
- **dih**: This example starts Solr in standalone mode with the DataImportHandler (DIH) enabled and several example `dataconfig.xml` files pre-configured for different types of data supported with DIH (such as, database contents, email, RSS feeds, etc.). The configset used is customized for DIH, and is found in `$SOLR_HOME/example/example-DIH/solr/conf`. For more information about DIH, see the section Uploading Structured Data Store Data with the Data Import Handler.
- **schemaless**: This example starts Solr in standalone mode using a managed schema, as described in the section Schema Factory Definition in SolrConfig, and provides a very minimal pre-defined schema. Solr will run in Schemaless Mode with this configuration, where Solr will create fields in the schema on the fly and will guess field types used in incoming documents. The configset used can be found in `$SOLR_HOME/server/solr/configsets/data_driven_schema_configs`.



The run in-foreground option (-f) does not work with the -e option since the script needs to perform additional tasks after starting the Solr server.

Stop

The stop command sends a STOP request to a running Solr node, which allows it to shutdown gracefully. The command will wait up to 5 seconds for Solr to stop gracefully and then will forcefully kill the process (kill -9).

`bin/solr stop [options]`

`bin/solr stop -help`

Available Parameters

Parameter	Description	Example
-p <port>	Stop Solr running on the given port. If you are running more than one instance, or are running in SolrCloud mode, you either need to specify the ports in separate requests or use the -all option.	`bin/solr stop -p 8983`
-all	Stop all running Solr instances that have a valid PID.	`bin/solr stop -all`
-k <key>	Stop key used to protect from stopping Solr inadvertently; default is "solrrocks".	`bin/solr stop -k solrrocks`

Informational

Version

The version command simply returns the version of Solr currently installed and immediately exists.

```
$ bin/solr version
X.Y.0
```

Status

The status command displays basic JSON-formatted information for any Solr nodes found running on the local system. The status command uses the SOLR_PID_DIR environment variable to locate Solr process ID files to find running Solr instances; the SOLR_PID_DIR variable defaults to the bin directory.

`bin/solr status`

The output will include a status of each node of the cluster, as in this example:

```
Found 2 Solr nodes:

Solr process 39920 running on port 7574
{
  "solr_home":"/Applications/Solr/example/cloud/node2/solr/",
  "version":"X.Y.0",
  "startTime":"2015-02-10T17:19:54.739Z",
  "uptime":"1 days, 23 hours, 55 minutes, 48 seconds",
  "memory":"77.2 MB (%15.7) of 490.7 MB",
  "cloud":{
    "ZooKeeper":"localhost:9865",
    "liveNodes":"2",
    "collections":"2"}}

Solr process 39827 running on port 8865
{
  "solr_home":"/Applications/Solr/example/cloud/node1/solr/",
  "version":"X.Y.0",
  "startTime":"2015-02-10T17:19:49.057Z",
  "uptime":"1 days, 23 hours, 55 minutes, 54 seconds",
  "memory":"94.2 MB (%19.2) of 490.7 MB",
  "cloud":{
    "ZooKeeper":"localhost:9865",
    "liveNodes":"2",
    "collections":"2"}}
```

Healthcheck

The healthcheck command generates a JSON-formatted health report for a collection when running in SolrCloud mode. The health report provides information about the state of every replica for all shards in a collection, including the number of committed documents and its current state.

`bin/solr healthcheck [options]`

`bin/solr healthcheck -help`

Available Parameters

Parameter	Description	Example
-c <collection>	Name of the collection to run a healthcheck against (required).	`bin/solr healthcheck -c gettingstarted`
-z <zkhost>	ZooKeeper connection string, defaults to localhost:9983. If you are running Solr on a port other than 8983, you will have to specify the ZooKeeper connection string. By default, this will be the Solr port + 1000.	`bin/solr healthcheck -z localhost:2181`

Below is an example healthcheck request and response using a non-standard ZooKeeper connect string, with 2 nodes running:

```
$ bin/solr healthcheck -c gettingstarted -z localhost:9865

{
  "collection":"gettingstarted",
  "status":"healthy",
  "numDocs":0,
  "numShards":2,
  "shards":[
    {
      "shard":"shard1",
      "status":"healthy",
      "replicas":[
        {
          "name":"core_node1",
          "url":"http://10.0.1.10:8865/solr/gettingstarted_shard1_replica2/",
          "numDocs":0,
          "status":"active",
          "uptime":"2 days, 1 hours, 18 minutes, 48 seconds",
          "memory":"25.6 MB (%5.2) of 490.7 MB",
          "leader":true},
        {
          "name":"core_node4",
          "url":"http://10.0.1.10:7574/solr/gettingstarted_shard1_replica1/",
          "numDocs":0,
          "status":"active",
          "uptime":"2 days, 1 hours, 18 minutes, 42 seconds",
          "memory":"95.3 MB (%19.4) of 490.7 MB"}]},
    {
      "shard":"shard2",
      "status":"healthy",
      "replicas":[
        {
          "name":"core_node2",
          "url":"http://10.0.1.10:8865/solr/gettingstarted_shard2_replica2/",
          "numDocs":0,
          "status":"active",
          "uptime":"2 days, 1 hours, 18 minutes, 48 seconds",
          "memory":"25.8 MB (%5.3) of 490.7 MB"},
        {
          "name":"core_node3",
          "url":"http://10.0.1.10:7574/solr/gettingstarted_shard2_replica1/",
          "numDocs":0,
          "status":"active",
          "uptime":"2 days, 1 hours, 18 minutes, 42 seconds",
          "memory":"95.4 MB (%19.4) of 490.7 MB",
          "leader":true}]}]}
```

Collections and Cores

The bin/solr script can also help you create new collections (in SolrCloud mode) or cores (in standalone mode), or delete collections.

Create

User permissions on "create"
When using the "create" command, be sure that you run this command as the same user that you use to start Solr. If you use the UNIX/Linux install script, this will normally be a user named "solr". If Solr is running as the solr user but you use root to create a core, then Solr will not be able to write to the directories created by the start script.

If you are running in cloud mode, this very likely will not be a problem. In cloud mode, all the configuration is stored in ZooKeeper, and the create script does not need to make directories or copy configuration files. Solr itself will create all the necessary directories.

The create command detects the mode that Solr is running in (standalone or SolrCloud) and then creates a core or collection depending on the mode.

`bin/solr create options`

`bin/solr create -help`

Available Parameters

Parameter	Description	Example
-c <name>	Name of the core or collection to create (required).	`bin/solr create -c mycollection`
-d <confdir>	The configuration directory. This defaults to `data_driven_schema_configs`. See the section Configuration Directories and SolrCloud below for more details about this option when running in SolrCloud mode.	`bin/solr create -d basic_configs`
-n <configName>	The configuration name. This defaults to the same name as the core or collection.	`bin/solr create -n basic`
-p <port>	Port of a local Solr instance to send the create command to; by default the script tries to detect the port by looking for running Solr instances. This option is useful if you are running multiple standalone Solr instances on the same host, thus requiring you to be specific about which instance to create the core in.	`bin/solr create -p 8983`
-s <shards> -shards	Number of shards to split a collection into, default is 1; only applies when Solr is running in SolrCloud mode.	`bin/solr create -s 2`
-rf <replicas> -replicationFactor	Number of copies of each document in the collection. The default is 1 (no replication).	`bin/solr create -rf 2`

Configuration Directories and SolrCloud

Before creating a collection in SolrCloud, the configuration directory used by the collection must be uploaded to ZooKeeper. The create command supports several use cases for how collections and configuration directories work. The main decision you need to make is whether a configuration directory in ZooKeeper should be shared across multiple collections. Let's work through a few examples to illustrate how configuration directories work in SolrCloud.

First, if you don't provide the `-d` or `-n` options, then the default configuration (`$SOLR_HOME/server/solr/configsets/data_driven_schema_configs/conf`) is uploaded to ZooKeeper using the same name as the

collection. For example, the following command will result in the **data_driven_schema_configs** configuration being uploaded to `/configs/contacts` in ZooKeeper: `bin/solr create -c contacts`. If you create another collection, by doing `bin/solr create -c contacts2`, then another copy of the `data_driven_schema_configs` directory will be uploaded to ZooKeeper under `/configs/contacts2`. Any changes you make to the configuration for the contacts collection will not affect the contacts2 collection. Put simply, the default behavior creates a unique copy of the configuration directory for each collection you create.

You can override the name given to the configuration directory in ZooKeeper by using the `-n` option. For instance, the command `bin/solr create -c logs -d basic_configs -n basic` will upload the `server/solr/configsets/basic_configs/conf` directory to ZooKeeper as `/configs/basic`.

Notice that we used the `-d` option to specify a different configuration than the default. Solr provides several built-in configurations under `server/solr/configsets`. However you can also provide the path to your own configuration directory using the `-d` option. For instance, the command `bin/solr create -c mycoll -d /tmp/myconfigs`, will upload `/tmp/myconfigs` into ZooKeeper under `/configs/mycoll` . To reiterate, the configuration directory is named after the collection unless you override it using the `-n` option.

Other collections can share the same configuration by specifying the name of the shared configuration using the `-n` option. For instance, the following command will create a new collection that shares the basic configuration created previously: `bin/solr create -c logs2 -n basic`.

Data-driven schema and shared configurations

The data_driven_schema_configs schema can mutate as data is indexed. Consequently, we recommend that you do not share data-driven configurations between collections unless you are certain that all collections should inherit the changes made when indexing data into one of the collections.

Delete

The delete command detects the mode that Solr is running in (standalone or SolrCloud) and then deletes the specified core (standalone) or collection (SolrCloud) as appropriate.

```
bin/solr delete [options]
```

```
bin/solr delete -help
```

If running in SolrCloud mode, the delete command checks if the configuration directory used by the collection you are deleting is being used by other collections. If not, then the configuration directory is also deleted from ZooKeeper. For example, if you created a collection by doing `bin/solr create -c contacts`, then the delete command `bin/solr delete -c contacts` will check to see if the `/configs/contacts` configuration directory is being used by any other collections. If not, then the `/configs/contacts` directory is removed from ZooKeeper.

Available Parameters

Parameter	Description	Example
-c <name>	Name of the core / collection to delete (required).	`bin/solr delete -c mycoll`
-deleteConfig <true|false>	Delete the configuration directory from ZooKeeper. The default is true. If the configuration directory is being used by another collection, then it will not be deleted even if you pass -deleteConfig true.	`bin/solr delete -deleteConfig false`

-p <port>	The port of a local Solr instance to send the delete command to. By default the script tries to detect the port by looking for running Solr instances. This option is useful if you are running multiple standalone Solr instances on the same host, thus requiring you to be specific about which instance to delete the core from.	`bin/solr delete -p 8983`

ZooKeeper Operations

The bin/solr script allows certain operations affecting Zookeeper. These operations are for SolrCloud mode only.

`bin/solr zk [options]`

`bin/solr zk -help`

NOTE: Solr should have been started at least once before issuing these commands to initialize Zookeeper with the znodes Solr expects. Once ZooKeeper is initialized, Solr doesn't need to be running on any node to use these commands.

Uploading a Configuration Set

Use this Zookeeper sub-command to upload one of the pre-configured configuration set or a customized configuration set to Zookeeper.

Available Parameters (all parameters are required)

Parameter	Description	Example
upconfig	Upload a configuration set from the local filesystem to Zookeeper (-upconfig for back compat)	`upconfig`
-n <name>	Name of the configuration set in Zookeeper. This command will upload the configuration set to the "configs" Zookeeper node giving it the name specified. You can see all uploaded configuration sets in the Admin UI via the Cloud screens. Choose Cloud->tree->configs to see them. If a pre-existing configuration set is specified, it will be overwritten in Zookeeper.	`-n myconfig`
-d <configset dir>	The path of the configuration set to upload. It should have a "conf" directory immediately below it that in turn contains solrconfig.xml etc. If just a name is supplied, $SOLR_HOME/server/solr/configsets will be checked for this name. An absolute path may be supplied instead.	`-d directory_under_configsets` `-d /absolute/path/to/configset/source`

-z <zkHost>	The Zookeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command with these parameters is:

```
bin/solr zk upconfig -z 111.222.333.444:2181 -n mynewconfig -d /path/to/configset
```

This command does **not** automatically make changes effective! It simply uploads the configuration sets to Zookeeper. You can use the Collections API to issue a RELOAD command for any collections that uses this configuration set.

Downloading a Configuration Set

Use this Zookeeper sub-command to download a configuration set from Zookeeper to the local filesystem.

Available Parameters (all parameters are required)

Parameter	Description	Example
downconfig	Download a configuration set from Zookeeper to the local filesystem. (-downconfig for back compat)	`downconfig`
-n <name>	Name of config set in Zookeeper to download. The Admin UI>>Cloud>>tree>>configs node lists all available configuration sets.	`-n myconfig`
-d <configset dir>	The path to write the downloaded configuration set into. If just a name is supplied, SOLR_HOME/server/solr/configsets will be the parent. An absolute path may be supplied as well. In either case, *pre-existing configurations at the destination will be overwritten!*	`-d directory_under_configsets -d /absolute/path/to/configset/destination`
-z <zkHost>	The Zookeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command with the parameters is:

```
bin/solr zk downconfig -z 111.222.333.444:2181 -n mynewconfig -d /path/to/configset
```

A "best practice" is to keep your configuration sets in some form of version control as the system-of-record. In that scenario, `downconfig` should rarely be used.

Copy between local files and Zookeeper znodes

Use this Zookeeper sub-command for transferring files and directories between Zookeeper znodes and your local drive. This command will copy from the local drive to Zookeeper, from Zookeeper to the local drive or from Zookeeper to Zookeeper.

Available Parameters

Parameter	Description	Example
cp	Copy files and directories to/from Zookeeper and the local drive	`cp`
-r	Optional. Do a recursive copy. The command will fail if the <src> has children unless '-r' is specified	`-r`
<src>	The file or path to copy from. If prefixed with 'zk:' then the source is presumed to be Zookeeper. If no prefix or the prefix is 'file:' this is the local drive. At least one of <src> or <dest> must be prefixed by 'zk:' or the command will fail.	`zk:/configs/myconfigs/solrconfig.xml` `file:/Users/apache/configs/src`
<dest>	The file or path to copy to. If prefixed with 'zk:' then the source is presumed to be Zookeeper. If no prefix or the prefix is 'file:' this is the local drive. At least one of <src> or <dest> must be prefixed by 'zk:' or the command will fail. If <dest> ends in a shash character it names a directory.	`zk:/configs/myconfigs/solrconfig.xml` `file:/Users/apache/configs/src`
-z <zkHost>	The ZooKeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command with the parameters is:

Recursively copy a directory from local to Zookeeper.

```
bin/solr zk cp -r  file:/apache/confgs/whatever/conf zk:/configs/myconf -z
111.222.333.444:2181
```

Copy a single file from Zookeeper to local.

```
bin/solr zk cp zk:/configs/myconf/managed_schema /configs/myconf/managed_schema -z
111.222.333.444:2181
```

Remove a znode from Zookeeper

Use this ZooKeeper sub-command to remove a znode (and optionally all child nodes) from Zookeeper

Available Parameters

Parameter	Description	Example
rm	Remove znode(s) from Zookeeper	`rm`
-r	Optional. Do a recursive removal. The command will fail if the <path> has children unless '-r' is specified	`-r`

<path>	The path to remove from Zookeeper, either a parent or leaf node There are limited safety checks, you cannot remove '/' or '/zookeeper' nodes. The path is assumed to be a Zookeeper node no 'zk:' prefix is necessary	/configs /configs/myconfigset /config/myconfigset/solrconfig.xml
-z <zkHost>	The ZooKeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command with the parameters is:

```
bin/solr zk rm -r /configs
```

```
bin/solr zk rm /configs/myconfigset/schema.xml
```

Move one Zookeeper znode to another (rename)

Use this ZooKeeper sub-command to move (rename) a Zookeeper znode

Available Parameters

Parameter	Description	Example
mv	Move/rename a znode	`mv`
<src>	Znode to rename. The 'zk:' prefix is assumed	`/configs/oldconfigset`
<dest>	The new name of the znode. The 'zk:' prefix is assumed	`/configs/newconfigset`
-z <zkHost>	The ZooKeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command is:

```
bin/solr zk mv /configs/oldconfigset /configs/newconfigset
```

List a Zookeeper znode's children

Use this ZooKeeper sub-command to see the children of a znode.

Available Parameters

Parameter	Description	Example
ls	Print out the children (optionally recursively) of a znode	`ls`
-r	Optional. Recursively list all descendants of a znode	`-r`
<path>	The path on Zookeeper to list.	`/collections/mycollection`
-z <zkHost>	The ZooKeeper connection string. Unnecessary if ZK_HOST is defined in solr.in.sh or solr.in.cmd	`-z 123.321.23.43:2181`

An example of this command with the parameters is:

```
bin/solr zk ls -r /collections/mycollection
bin/solr zk ls /collections
```

Upgrading Solr

If you are already using Solr 6.1, Solr 6.2 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. Detailed steps for upgrading a Solr cluster can be found in the appendix: Upgrading a Solr Cluster.

Upgrading from 6.1.x

No major issues expected.

Upgrading from earlier 6.x versions

- If you use historical dates, specifically on or before the year 1582, you should re-index after upgrading to this version.

Upgrading from 5.5.x

- The deprecated `SolrServer` and subclasses have been removed, use `SolrClient` instead.
- The deprecated `<nrtMode>` configuration in `solrconfig.xml` has been removed. Please remove it from `solrconfig.xml`.
- `SolrClient.shutdown()` has been removed, use `SolrClient.close()` instead.
- The deprecated `zkCredientialsProvider` element in `solrcloud` section of `solr.xml` is now removed. Use the correct spelling (`zkCredentialsProvider`) instead.
- Internal/expert - `ResultContext` was significantly changed and expanded to allow for multiple full query results (`DocLists`) per Solr request. `TransformContext` was rendered redundant and was removed. See SOLR-7957 for details.
- Several changes have been made regarding the "`Similarity`" used in Solr, in order to provide better default behavior for new users. There are 3 key impacts of these changes on existing users who upgrade:
  - `DefaultSimilarityFactory` has been removed. If you currently have `DefaultSimilarityFactory` explicitly referenced in your `schema.xml`, edit your config to use the functionally identical `ClassicSimilarityFactory`. See SOLR-8239 for more details.
  - The implicit default Similarity used when no `<similarity/>` is configured in `schema.xml` has been changed to `SchemaSimilarityFactory`. Users who wish to preserve back-compatible behavior should either explicitly configure `ClassicSimilarityFactory`, or ensure that the `luceneMatchVersion` for the collection is less then 6.0. See SOLR-8270 + SOLR-8271 for details.
  - `SchemaSimilarityFactory` has been modified to use `BM25Similarity` as the default for `fieldTypes` that do not explicitly declare a Similarity. The legacy behavior of using `ClassicSimilarity` as the default will occur if the `luceneMatchVersion` for the collection is less then 6.0, or the '`defaultSimFromFieldType`' configuration option may be used to specify any default of your choosing. See SOLR-8261 + SOLR-8329 for more details.
- If your `solrconfig.xml` file doesn't explicitly mention the `schemaFactory` to use then Solr will choose the `ManagedIndexSchemaFactory` by default. Previously it would have chosen `ClassicIndexSchemaFactory`. This means that the Schema APIs (`/<collection>/schema`) are enabled and the schema is mutable. When Solr starts your `schema.xml` file will be renamed to `managed-schema`. If you want to retain the old behaviour then please ensure that the `solrconfig.xml` explicitly uses the `ClassicIndexSchemaFactory` or your `luceneMatchVersion` in the `solrconfig.xml` is less than 6.0. See the Schema Factory Definition in SolrConfig section for more details
- `SolrIndexSearcher.QueryCommand` and `QueryResult` were moved to their own classes. If you reference them in your code, you should import them under o.a.s.search (or use your IDE's "Organize Imports").

- The '`useParams`' attribute specified in request handler cannot be overridden from request params. See SOLR-8698 for more details.
- When requesting stats in date fields, "sum" is now returned as a double value instead of a date. See SOLR-8671 for more details.
- The deprecated GET methods for schema are now accessible through the bulk API. These methods now accept fewer request parameters, and output less information. See SOLR-8736 for more details. Some of the removed functionality will likely be restored in a future version of Solr - see SOLR-8992.
- In the past, Solr guaranteed that retrieval of multi-valued fields would preserve the order of values. Because values may now be retrieved from column-stored fields (`docValues="true"`), in conjunction with the fact that DocValues do not currently preserve order, means that users should set `useDocValuesAsStored="false"` to prevent future optimizations from using the column-stored values over the row-stored values when fields have both `stored="true"` and `docValues="true"`.
- Formatted date-times from Solr have some differences. If the year is more than 4 digits, there is a leading '+'. When there is a non-zero number of milliseconds, it is padded with zeros to 3 digits. Negative year (BC) dates are now possible. Parsing: It is now an error to supply a portion of the date out of its, range, like 67 seconds.
- SolrJ no longer includes `DateUtil`. If for some reason you need to format or parse dates, simply use `Instant.format()` and `Instant.parse()`.
- If you are using spatial4j, please upgrade to 0.6 and edit your `spatialContextFactory` to replace `com.spatial4j.core` with `org.locationtech.spatial4j`.

Upgrading from Older 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.

A summary of the significant changes between Solr 5.x and Solr 6.0 can be found in the Major Changes from Solr 5 to Solr 6 section.

Using the Solr Administration User Interface

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

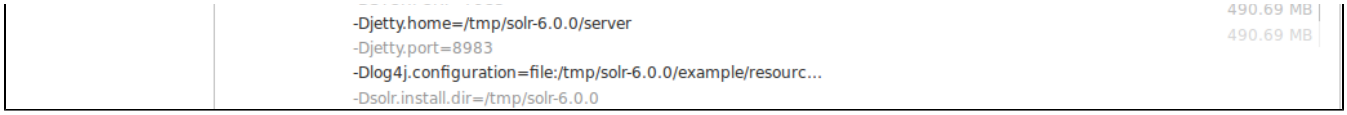
The Overview of the Solr Admin UI explains the basic features of the user interface, what's on the initial Admin UI page, and how to configure the interface. In addition, there are pages describing each screen of the Admin UI:

- **Getting Assistance** shows you how to get more information about the UI.
- **Logging** shows recent messages logged by this Solr node and provides a way to change logging levels for specific classes.
- **Cloud Screens** display information about nodes when running in SolrCloud mode.
- **Collections / 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.
- **Collection-Specific Tools** is a section explaining additional screens available for each collection.
    - 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`.
    - Query - lets you submit a structured query about various elements of a core.
    - Stream - allows you to submit streaming expressions and see results and parsing explanations.
    - Schema Browser - displays schema data in a browser window.
- **Core-Specific Tools** is a section explaining additional screens available for each named core.
    - Ping - lets you ping a named core and determine whether the core is active.
    - Plugins/Stats - shows statistics for plugins and other installed components.
    - Replication - shows you the current replication status for the core, and lets you enable/disable replication.
    - Segments Info - Provides a visualization of the underlying Lucene index segments.

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.





Accessing the URL `http://hostname:8983/solr/` 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, Collection/Core Administration and Java Properties, among other things. At the end of this information is at least one pulldown listing Solr cores configured for this instance. On SolrCloud nodes, an additional pulldown list shows all collections in this cluster. Clicking on a collection or core name shows secondary menus of information for the specified collection or core, such as a Schema Browser, Config Files, Plugins & Statistics, 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.

Under the covers, the Solr Admin UI re-uses the same HTTP APIs available to all clients to access Solr-related data to drive an external interface.

The path to the Solr Admin UI given above is `http://hostname:port/solr`, which redirects to `http://hostname:port/solr/#/` in the current version. A convenience redirect is also supported, so simply accessing the Admin UI at `http://hostname:port/` will also redirect to `http://hostname:port/solr/#/`.

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 https://lucene.apache.org/solr/.
Issue Tracker	Navigates to the JIRA issue tracking server for the Apache Solr project. This server resides at https://issues.apache.org/jira/browse/SOLR.
IRC Channel	Navigates to an Apache Wiki page describing how to join Solr's IRC live-chat room: https://wiki.apache.org/solr/IRCChannels.
Community forum	Navigates to the Apache Wiki page which has further information about ways to engage in the Solr User community mailing lists: https://wiki.apache.org/solr/UsingMailingLists.

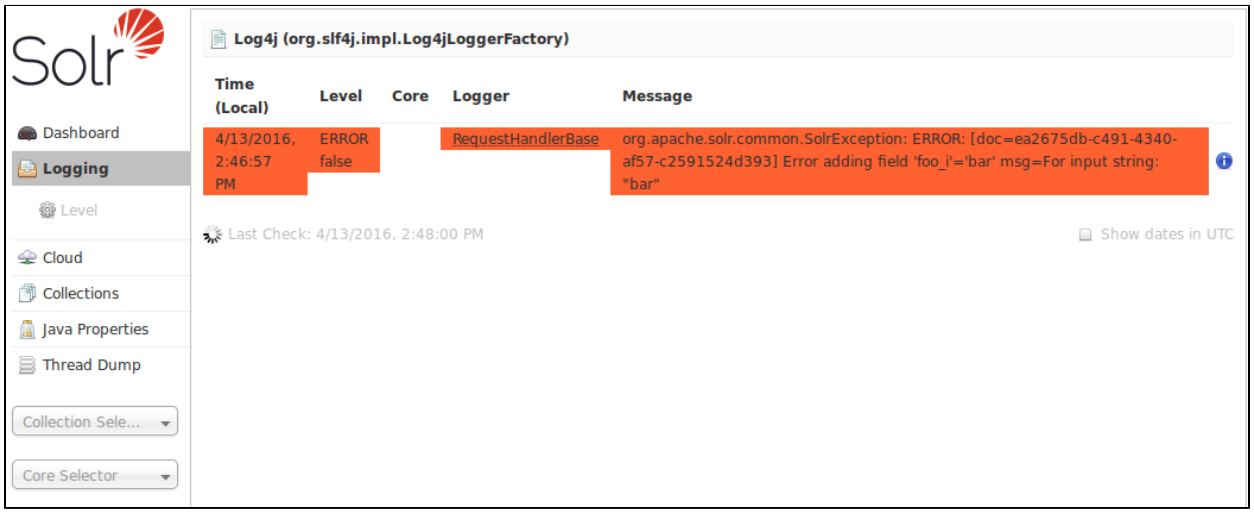
Solr Query Syntax	Navigates to the section "Query Syntax and Parsing" in this reference guide.

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 recent messages logged by this Solr node.

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

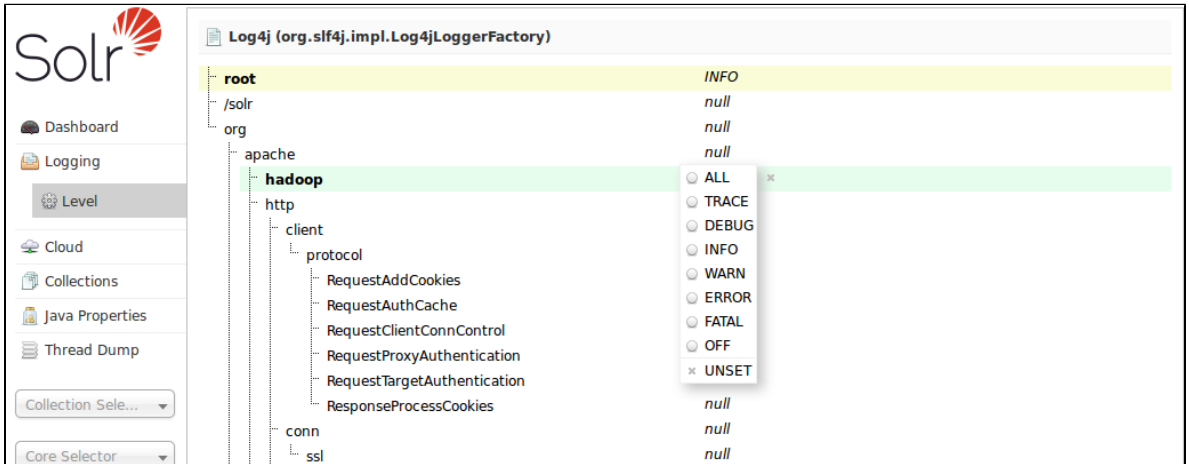


The Main Logging Screen, including an example of an error due to a bad document sent by a client

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

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.

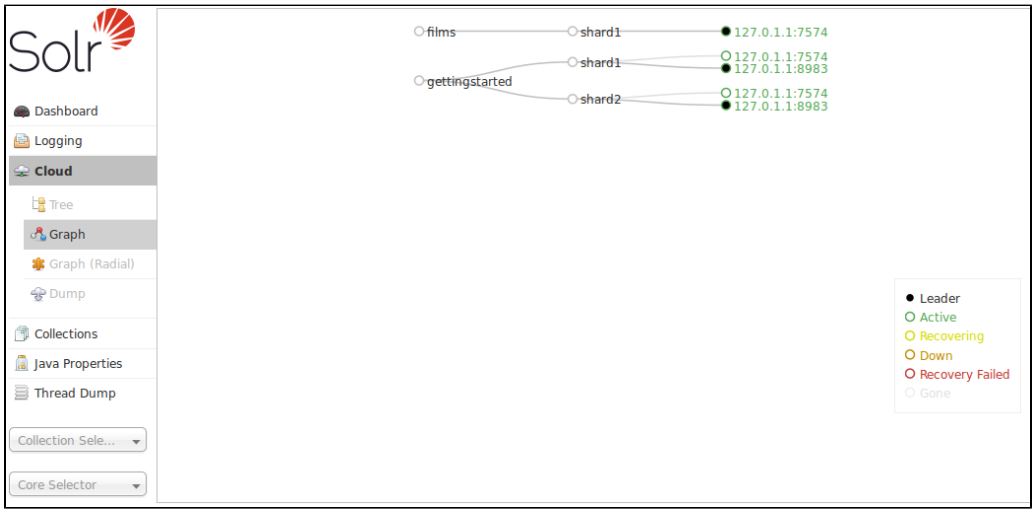
Cloud Screens

When running in SolrCloud mode, a "Cloud" option will appear in the Admin UI between Logging and Collections/Core Admin which provides status information about each collection & node in your cluster, as well as access to the low level data being stored in Zookeeper.

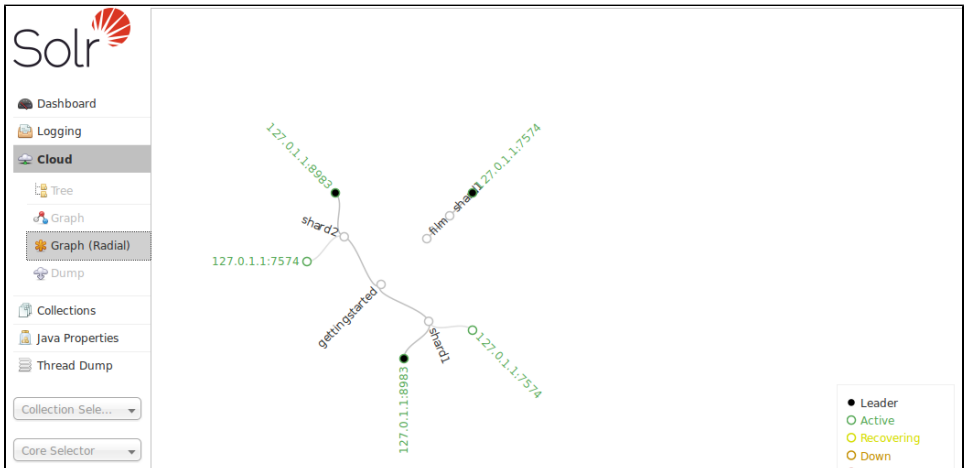
> **Only Visible When using SolrCloud**
> The "Cloud" menu option is only available on Solr instances running in SolrCloud mode. Single node or master/slave replication instances of Solr will not display this option.

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 ("Graph") shows a graph of each collection, the shards that make up those collections, and the addresses of each replica for each shard.

This example shows the very simple two-node cluster created using the "`bin/solr -e cloud -noprompt`" example command. In addition to the 2 shard, 2 replica "gettingstarted" collection, there is an additional "films" collection consisting of a single shard/replica:

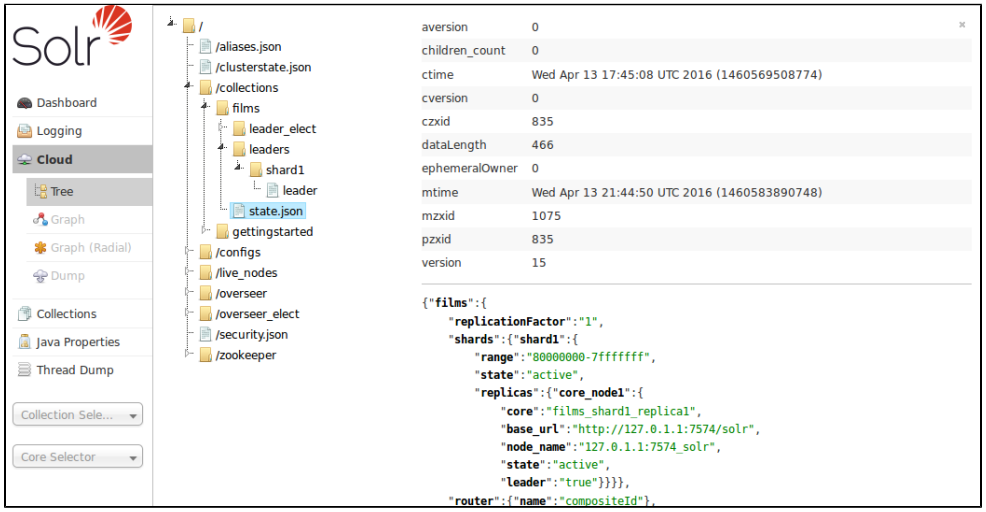


The "Graph (Radial)" option provides a different visual view of each node. Using the same example cluster, the radial graph view looks like:





The "Tree" option shows a directory structure of the data in ZooKeeper, including cluster wide information regarding the `live_nodes` and `overseer` status, as well as collection specific information such as the `state.json`, current shard leaders, and configuration files in use. In this example, we see the `state.json` file definition for the "films" collection:



The final option is "Dump", which returns a JSON document containing all nodes, their contents and their children (recursively). This can be used to export a snapshot of all the data that Solr has kept inside ZooKeeper and can aid in debugging SolrCloud problems.

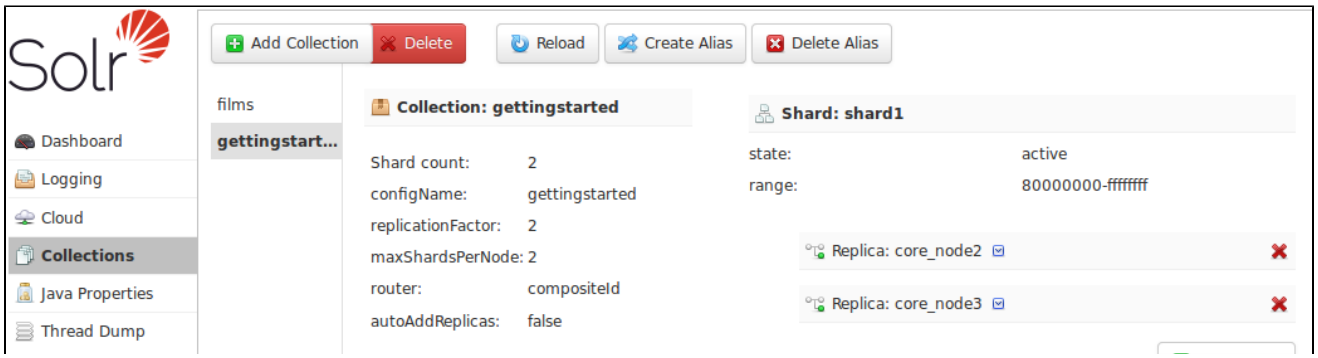
Collections / Core Admin

The Collections screen provides some basic functionality for managing your Collections, powered by the Collections API.

> If you are running a single node Solr instance, you will not see a Collections option in the left nav menu of the Admin UI.
>
> You will instead see a "Core Admin" screen that supports some comparable Core level information & manipulation via the CoreAdmin API instead.

The main display of this page provides a list of collections that exist in your cluster. Clicking on a collection name provides some basic metadata about how the collection is defined, and it's current shards & replicas, with options for adding and deleting individual replicas.

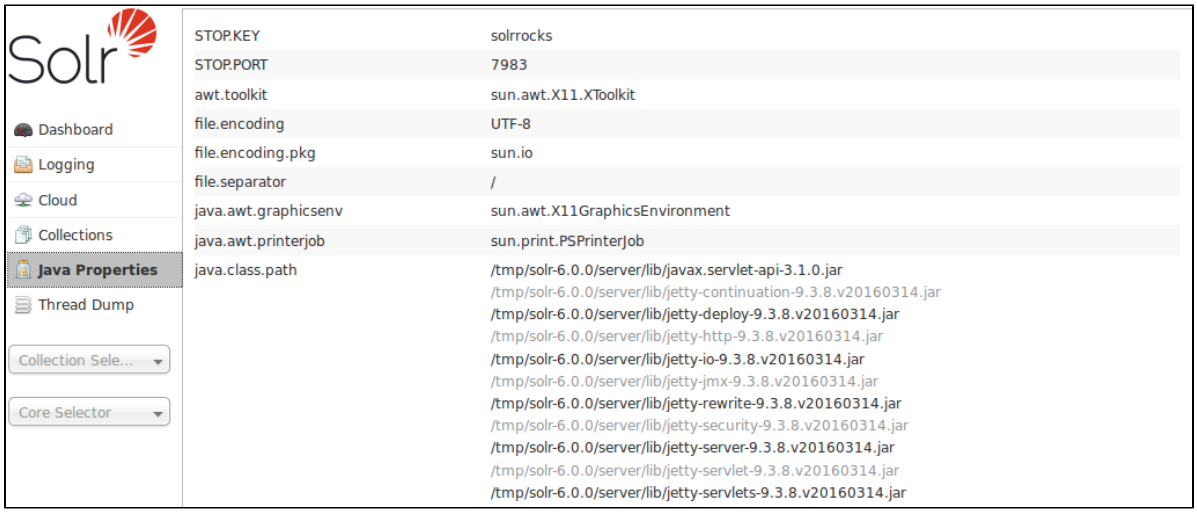
The buttons at the top of the screen let you make various collection level changes to your cluster, from add new collections or aliases to reloading or deleting a single collection.





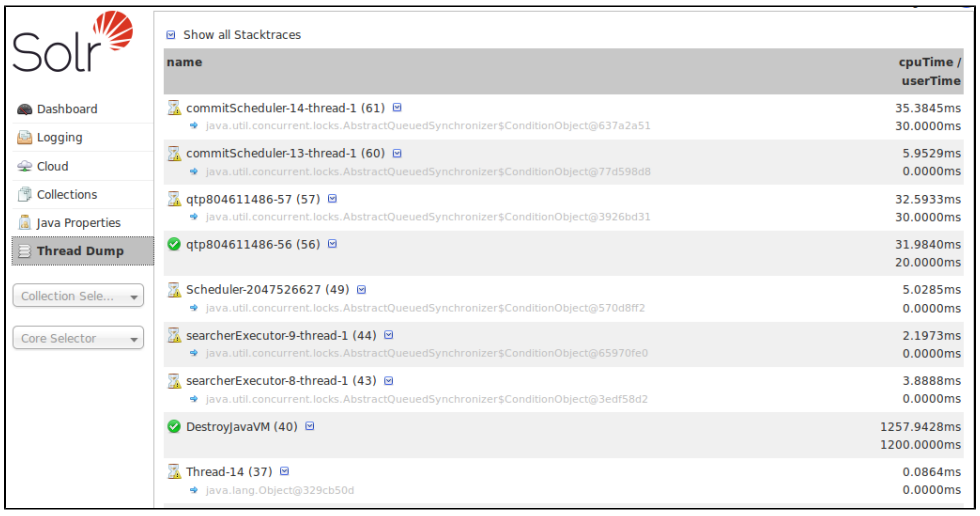
Java Properties

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.



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.

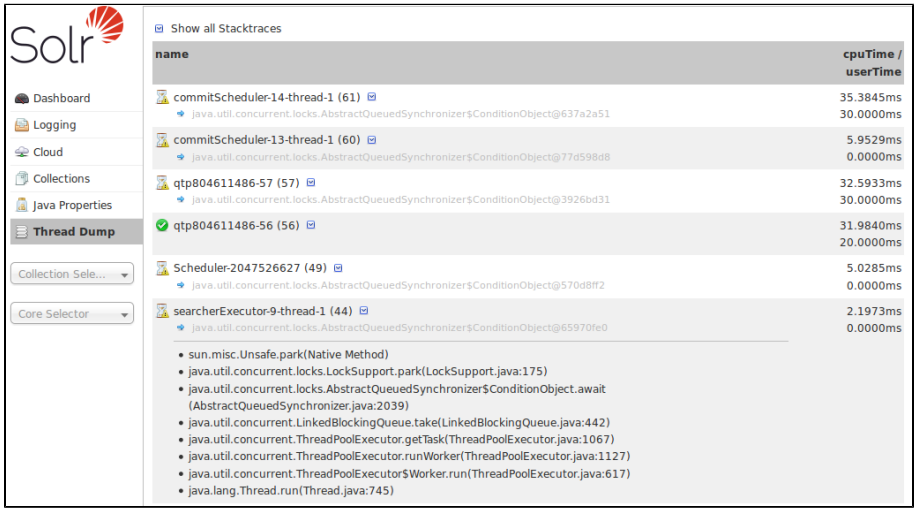


When you move your cursor over a thread name, a box floats over the name with the state for that thread. Thread states can be:

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:



Inspecting a thread

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

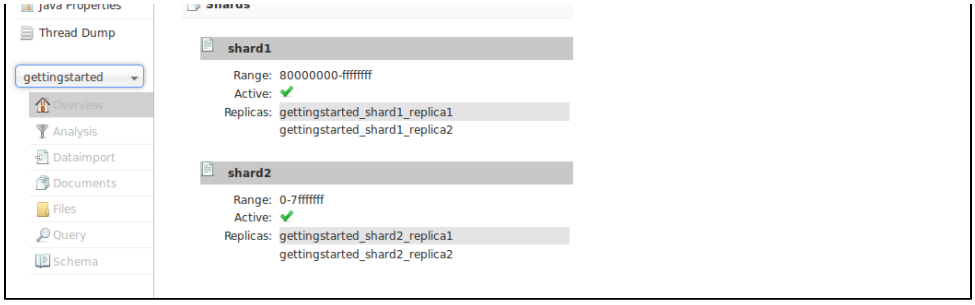
Collection-Specific Tools

In the left-hand navigation bar, you will see a pull-down menu titled "Collection Selector" that can be used to access collection specific administration screens.

> **Only Visible When using SolrCloud**
> The "Collection Selector" pull-down menu is only available on Solr instances running in SolrCloud mode.
>
> Single node or master/slave replication instances of Solr will not display this menu, instead the Collection specific UI pages described in this section will be available in the Core Selector pull-down menu.

Clicking on the Collection Selector pull-down menu will show a list of the collections in your Solr cluster, with a search box that can be used to find a specific collection by name. When you select a collection from the pull-down, the main display of the page will display some basic metadata about the collection, and a secondary menu will appear in the left nav with links to additional collection specific administration screens.





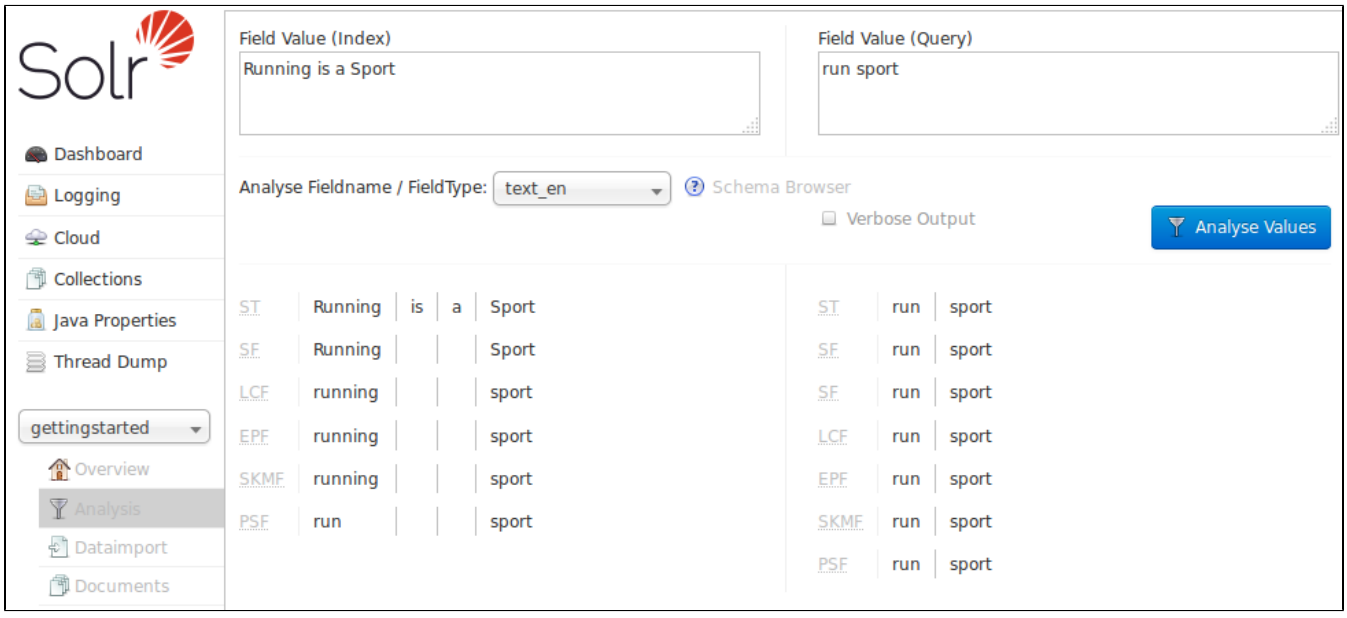
The collection-specific UI screens are listed below, with a link to the section of this guide to find out more:

- 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`.
- Query - lets you submit a structured query about various elements of a core.
- Stream - allows you to submit streaming expressions and see results and parsing explanations.
- Schema Browser - displays schema data in a browser window.

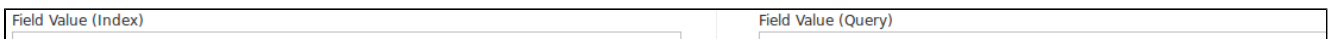
Analysis Screen

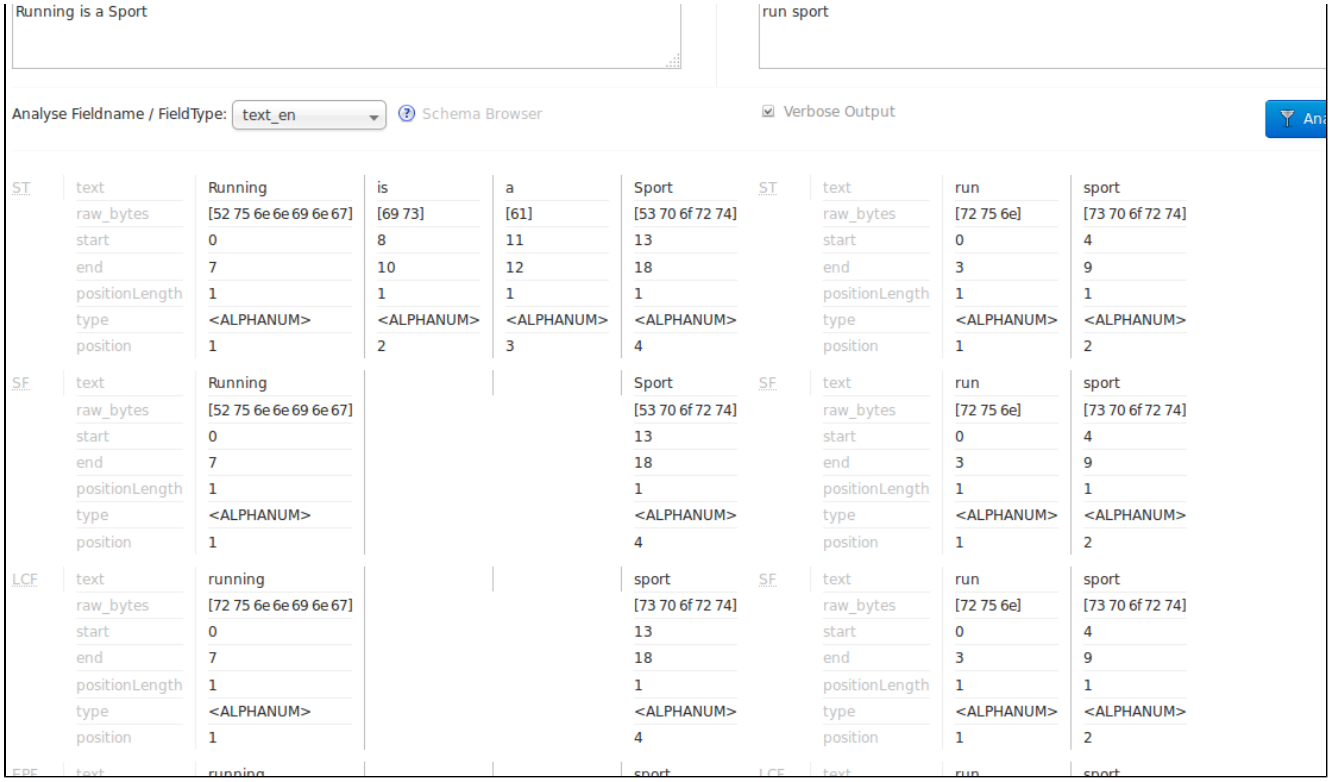
The Analysis screen lets you inspect how data will be handled according to the field, field type and dynamic field configurations found in your Schema. You can analyze how content would be handled during indexing or during query processing and view the results separately or at the same time. Ideally, 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.



If you click the **Verbose Output** check box, you see more information, including more details on the transformations to the input (such as, convert to lower case, strip extra characters, etc.) including the raw bytes, type and detailed position information at each stage. 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.



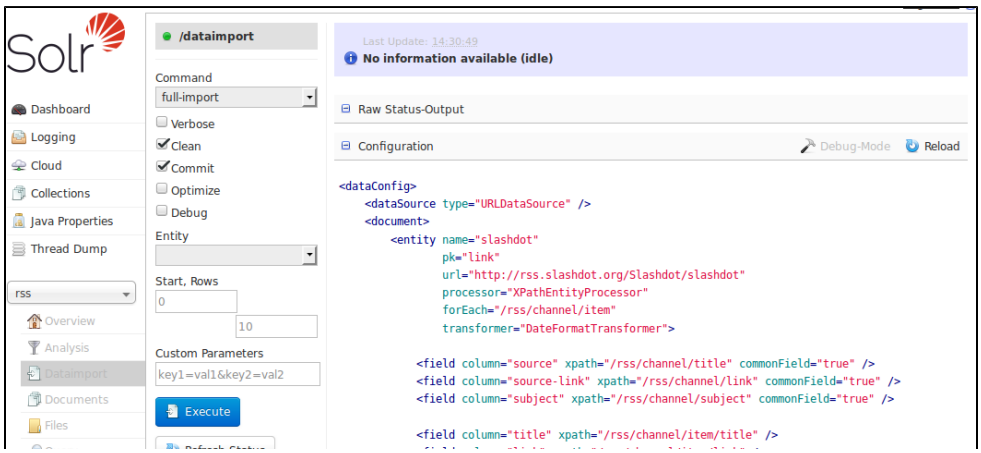


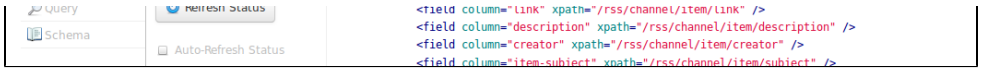
In the example screenshot above, several transformations are applied to the input "Running is a sport." The words "is" and "a" have been removed and the word "running" has been changed to its basic form, "run". This is because we are using the field type `text_en` in this scenario, which is configured 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 start, and monitor the status of, import commands as defined by the options selected on the screen and defined in the configuration file.

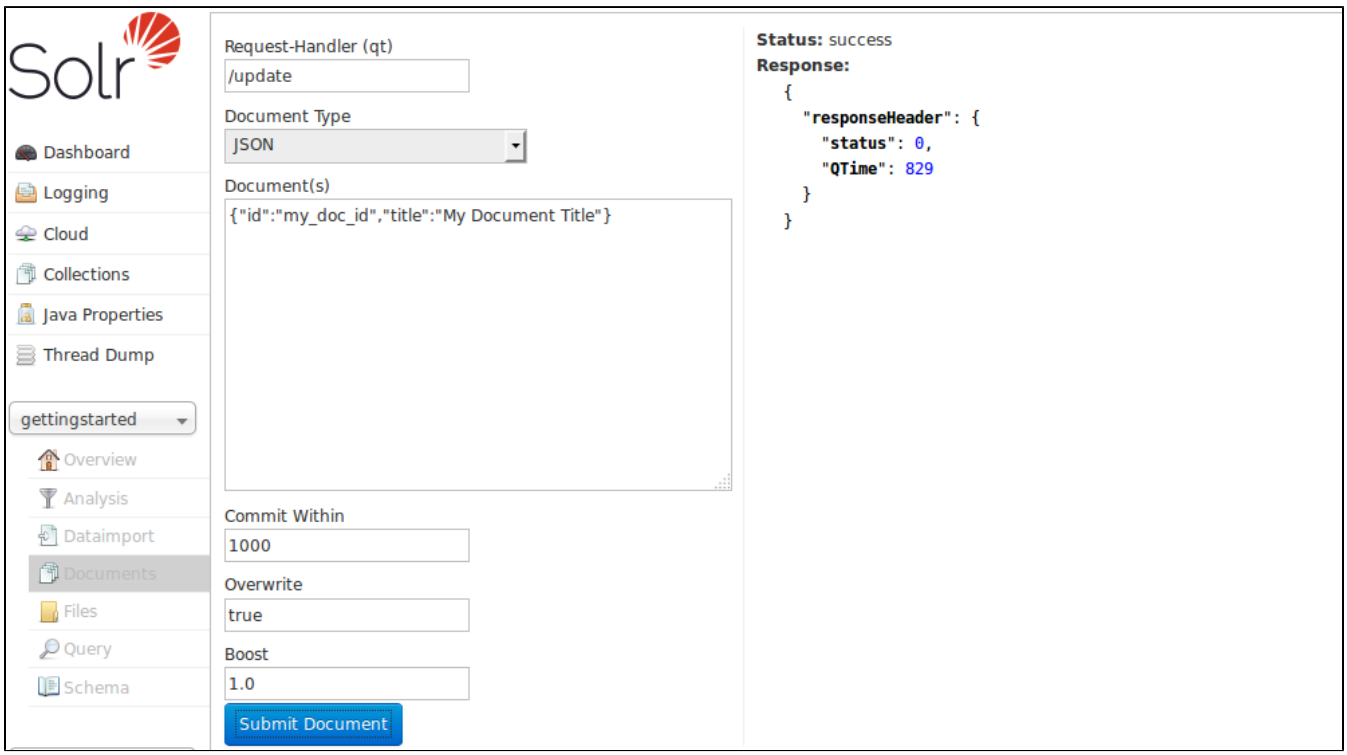




This screen also lets you adjust various options to control how the data is imported to Solr, and view the data import configuration file that controls the import. For more information about data importing with DIH, see the section on Uploading Structured Data Store Data with the Data Import Handler.

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.



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 opti on.

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).

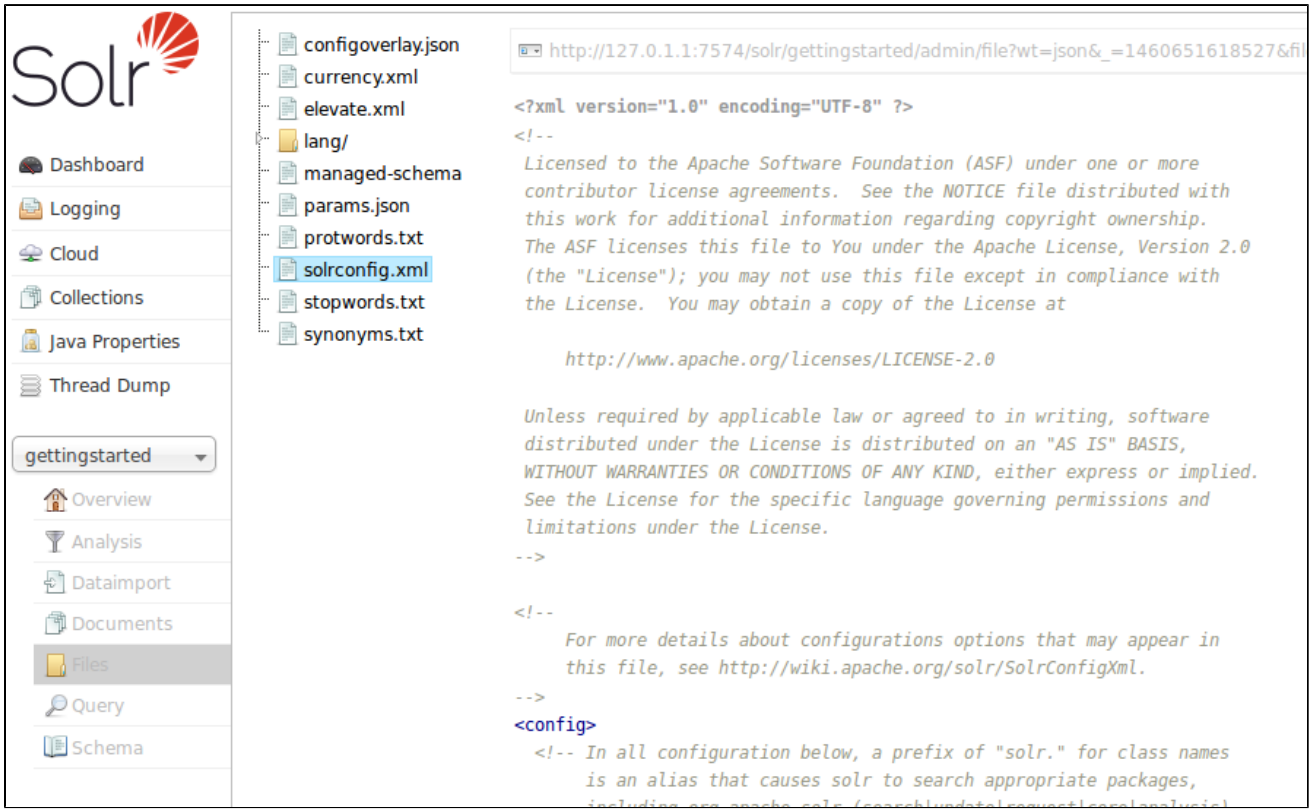
This option will only add or overwrite documents to the index; for other update tasks, see the Solr Command opti on.

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 the schema file) for the collection you selected.



If you are using SolrCloud, the files displayed are the configuration files for this collection stored in ZooKeeper (using `upconfig`), for single node Solr installations, all files in the `./conf` directory are displayed.

While `solrconfig.xml` defines the behaviour of Solr as it indexes content and responds to queries, the Schema 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 your schema.

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.

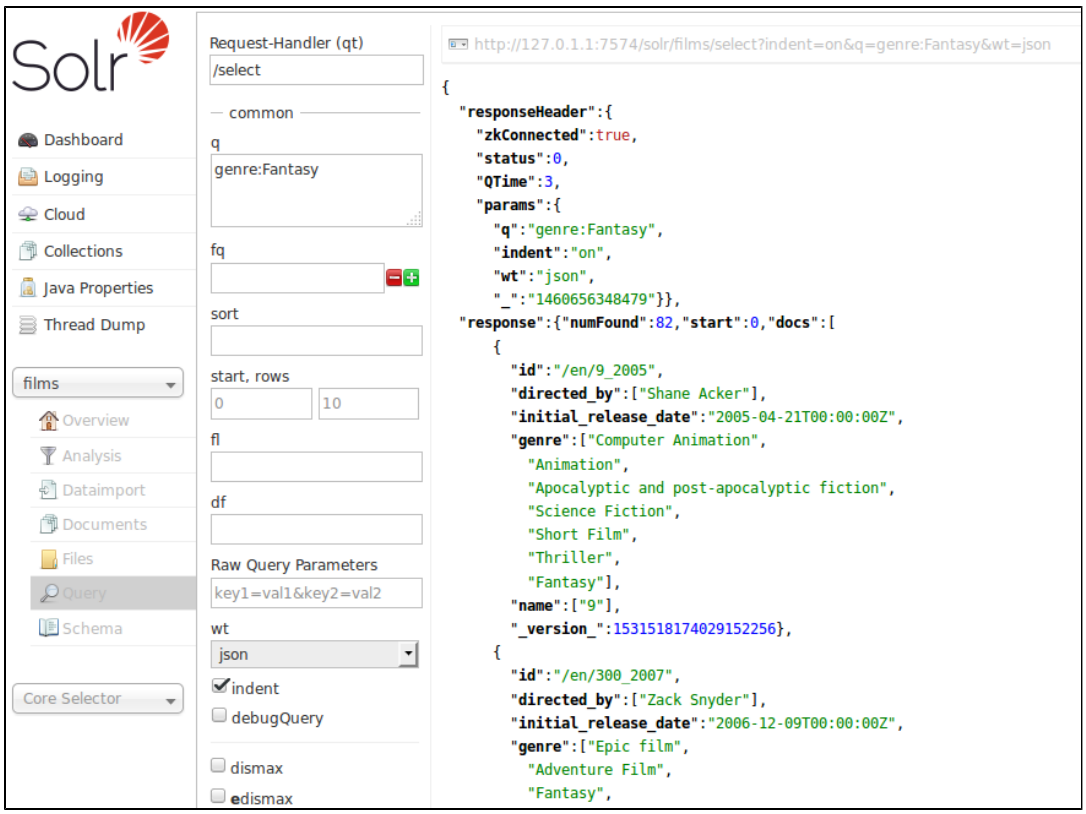
Many of the options defined in these configuration files are described throughout the rest of this Guide. In particular, you will want to review these sections:

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

Query Screen

You can use the **Query** screen to submit a search query to a Solr collection 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.



In this example a query for `genre:Fantasy` was sent to a "films" collection. Defaults were used for all other options in the form, which are explained briefly in the table below, and 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, functions, and doc transformers you want to have returned by separating them with either a comma or a space.
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

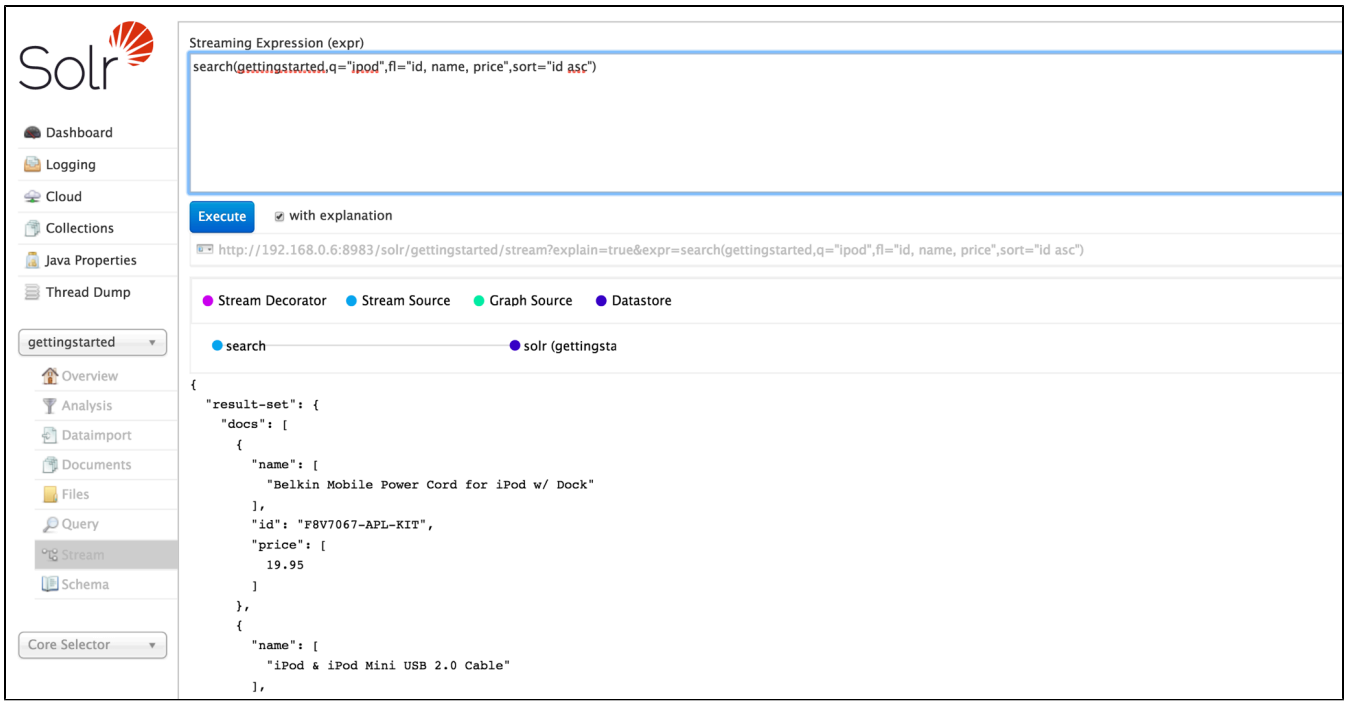
Stream Screen

The Stream screen allows you to enter a streaming expression and see the results. It is very similar to the Query Screen, except the input box is at the top and all options must be declared in the expression.

The screen will insert everything up to the streaming expression itself, so you do not need to enter the full URI with the hostname, port, collection, etc. Simply input the expression after the `expr=` part, and the URL will be constructed dynamically as appropriate.

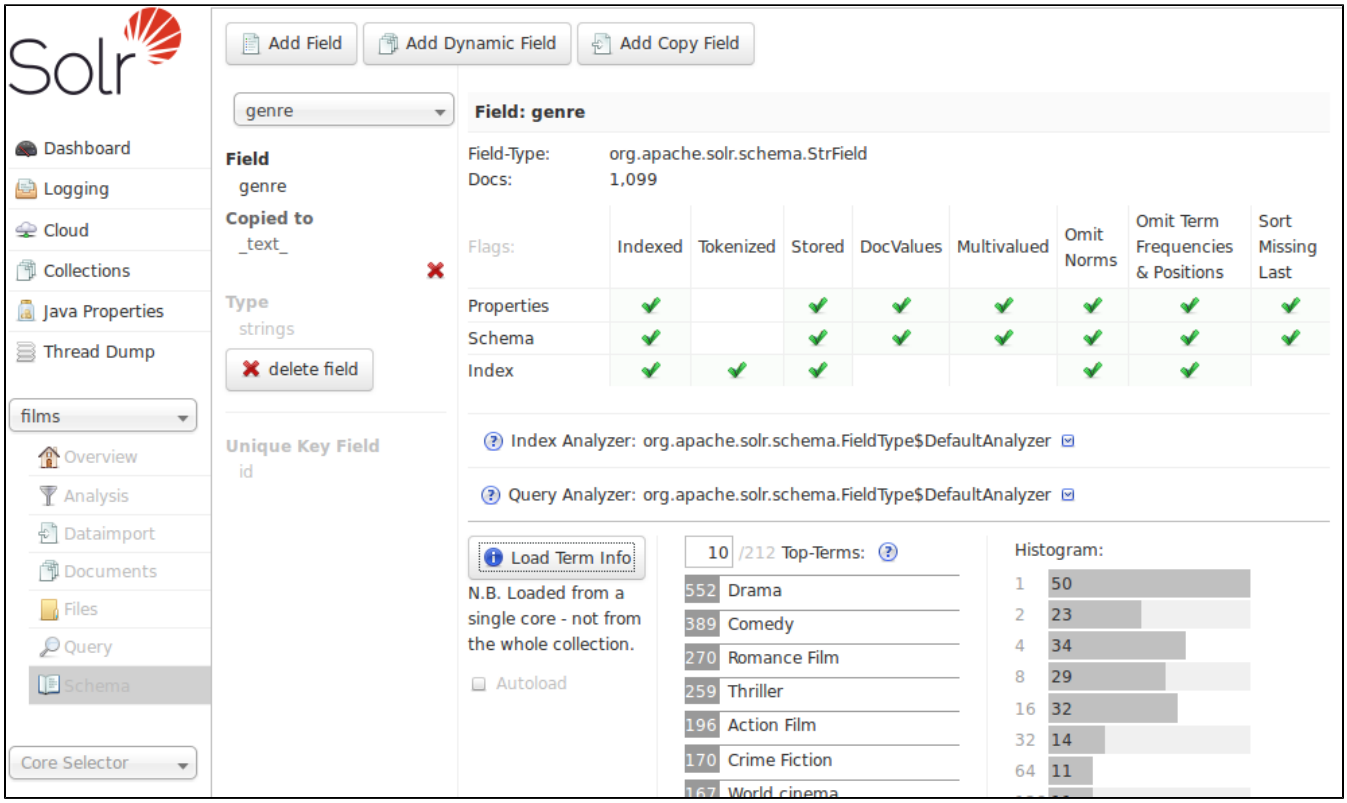
Under the input box, the Execute button will run the expression. An option "with explanation" will show the parts of the streaming expression that were executed. Under this, the streamed results are shown. A URL to be able to

view the output in a browser is also available.



Schema Browser Screen

The Schema Browser screen lets you review 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.



The screen provides a great deal of useful information about each particular field and fieldtype in the Schema, and provides a quick UI for adding fields or fieldtypes using the Schema API (if enabled). In the example above, we have chosen the `genre` field. On the left side of the main view window, we see the field name, that it is copied to the `_text_` (because of a copyField rule) and that it use the `strings` fieldtype. Click on one of those field or fieldtype names, and you can see the corresponding definitions.

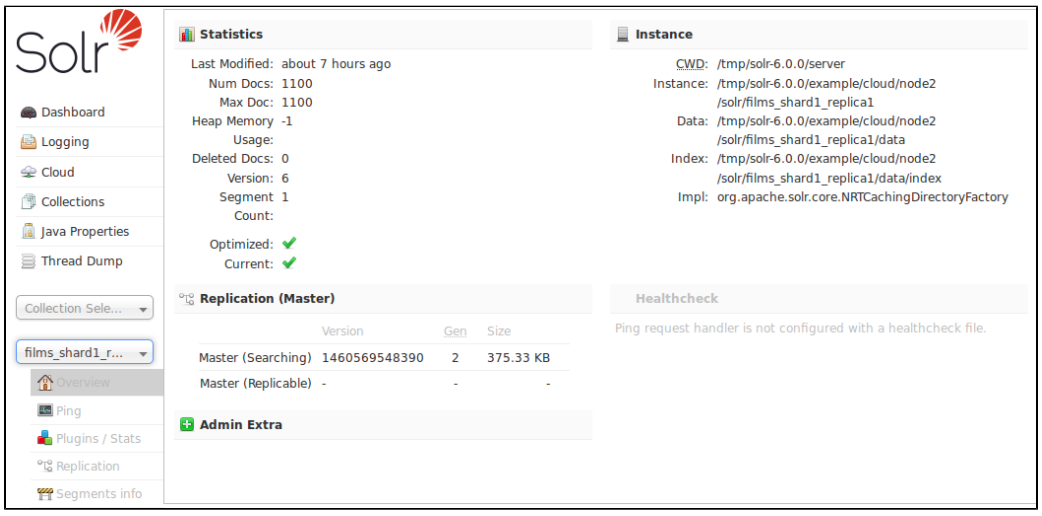
In the right part of the main view, we see the specific properties of how the `genre` field is defined – either explicitly or implicitly via it's fieldtype, as well as how many documents have populated this field. Then we see the analyzer used for indexing and query processing. Click the icon to the 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 a sample shard for that field, as well as a histogram showing the number of terms with various frequencies. 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.

Term Information is loaded from single arbitrarily selected core from the collection, to provide a representative sample for the collection. Full Field Facet query results are needed to see precise term counts across the entire collection.

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 hosted on this Solr node, with a search box that can be used to find a specific core by name. When you select a core from the pull-down, the main display of the page will display some basic metadata about the core, and a secondary menu will appear in the left nav with links to additional core specific administration screens. You can also define a configuration 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.



The core-specific UI screens are listed below, with a link to the section of this guide to find out more:

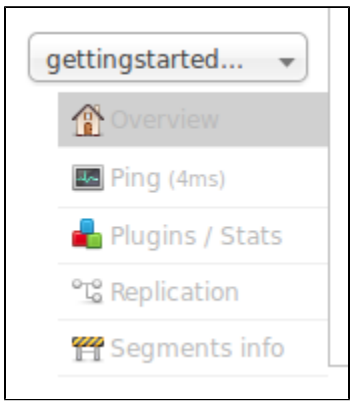
- Ping - lets you ping a named core and determine whether the core is active.
- Plugins/Stats - shows statistics for plugins and other installed components.
- Replication - shows you the current replication status for the core, and lets you enable/disable replication.
- Segments Info - Provides a visualization of the underlying Lucene index segments.

If you are running a single node instance of Solr, additional UI screens normally displayed on a per-collection bases will also be listed:

- 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`.
- Query - lets you submit a structured query about various elements of a core.
- Stream - allows you to submit streaming expressions and see results and parsing explanations.
- Schema Browser - displays schema data in a browser window.

Ping

Choosing Ping under a core name issues a `ping` request to check whether the core is up and responding to requests.



The search executed by a 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.

API Examples

While the UI screen makes it easy to see the ping response time, the underlying ping command can be more

useful when executed by remote monitoring tools:

Input

```
http://localhost:8983/solr/<core-name>/admin/ping
```

This command will ping the core name for a response.

Input

```
http://localhost:8983/solr/<collection-name>admin/ping?wt=json&distrib=true&indent=true
```

This command will ping all replicas of the given collection name for a response

Sample Output

```
<response>
   <lst name="responseHeader">
      <int name="status">0</int>
      <int name="QTime">13</int>
      <lst name="params">
         <str name="q">{!lucene}*:*</str>
         <str name="distrib">false</str>
         <str name="df">_text_</str>
         <str name="rows">10</str>
         <str name="echoParams">all</str>
      </lst>
   </lst>
   <str name="status">OK</str>
</response>
```

Both API calls have the same output. A status=OK indicates that the nodes are responding.

SolrJ Example

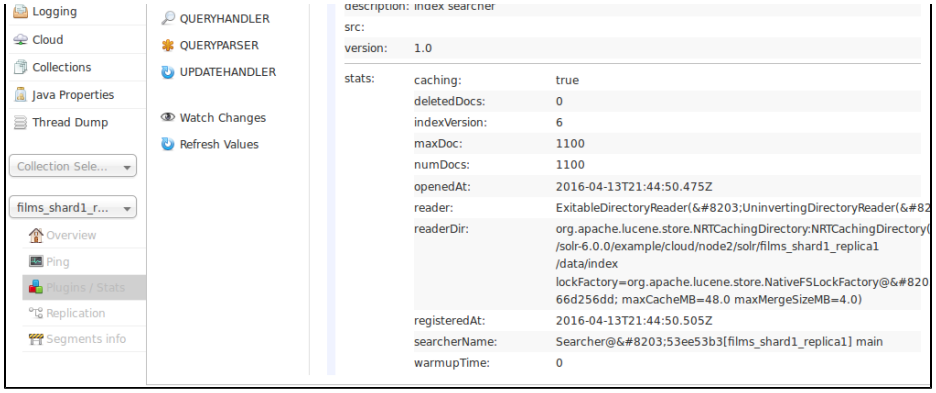
```
SolrPing ping = new SolrPing();
ping.getParams().add("distrib", "true"); //To make it a distributed request against a collection
rsp = ping.process(solrClient, collectionName);
int status = rsp.getStatus();
```

Plugins & Stats Screen

The Plugins screen shows information and statistics about the status and performance of various plugins running in each Solr core. You can find information about the performance of the Solr caches, the state of Solr's searchers, and the configuration of Request Handlers and Search Components.

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 Statistics

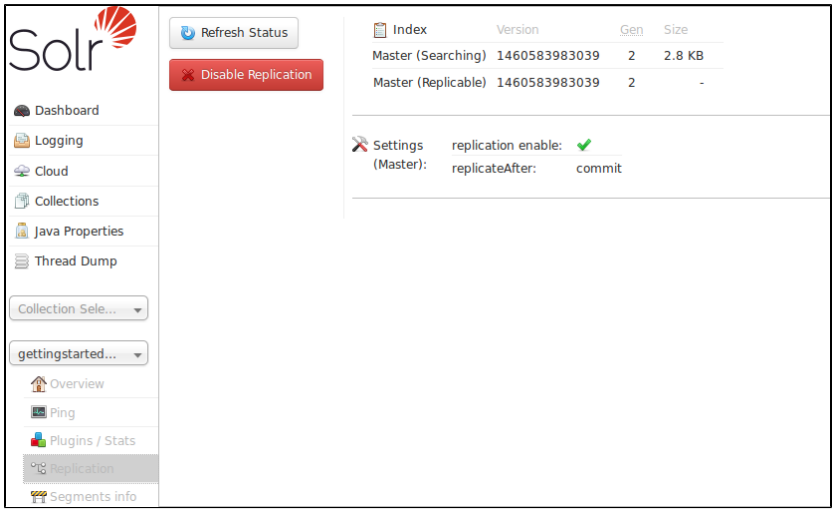
The display is a snapshot taken when the page is loaded. You can get updated status by choosing to either **Wat ch Changes** or **Refresh Values**. Watching the changes will highlight those areas that have changed, while refreshing the values will reload the page with updated information.

Replication Screen

The Replication screen shows you the current replication state for the core you have specified. SolrCloud has supplanted much of this functionality, but if you are still using Master-Slave index replication, you can use this screen to:

1. View the replicatable index state. (on a master node)
2. View the current replication status (on a slave node)
3. Disable replication. (on a master node)

Caution When Using SolrCloud
When using SolrCloud, do not attempt to disable replication via this screen.

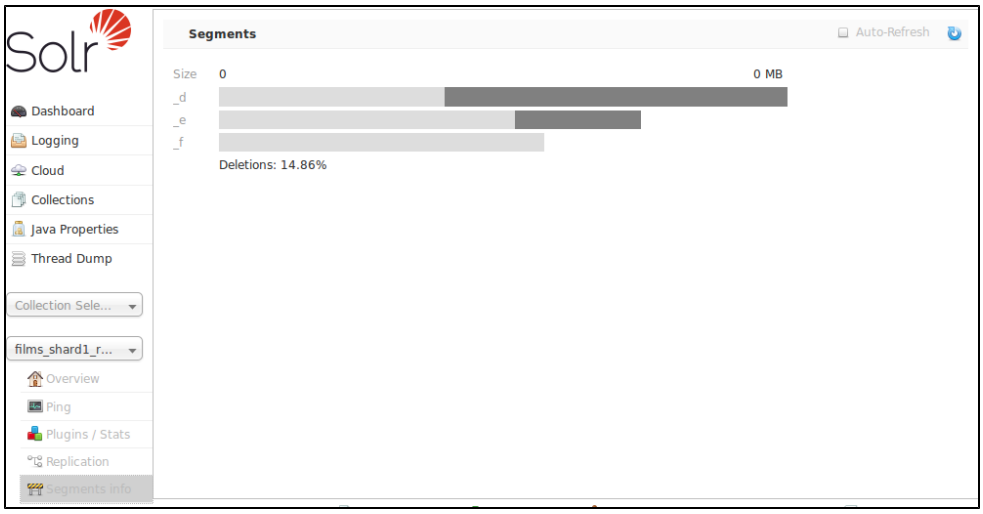


More details on how to configure replication is available in the section called Index Replication.

Segments Info

The Segments Info screen lets you see a visualization of the various segments in the underlying Lucene index for this core, with information about the size of each segment – both bytes and in number of documents – as well as other basic metadata about those segments. Most visible is the the number of deleted documents, but you

can hover your mouse over the segments to see additional numeric details.



This information may be useful for people to help make decisions about the optimal merge settings for their data.

Documents, Fields, and Schema Design

This section discusses how Solr organizes its data into documents and fields, as well as how to work with a schema in Solr.

This section 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.

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 *up dating*. 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's Schema File

Solr stores details about the field types and fields it is expected to understand in a schema file. The name and location of this file may vary depending on how you initially configured Solr or if you modified it later.

- `managed-schema` is the name for the schema file Solr uses by default to support making Schema changes at runtime via the Schema API, or Schemaless Mode features. You may explicitly configure the managed schema features to use an alternative filename if you choose, but the contents of the files are still updated automatically by Solr.
- `schema.xml` is the traditional name for a schema file which can be edited manually by users who use the `ClassicIndexSchemaFactory`.
- If you are using SolrCloud you may not be able to find any file by these names on the local filesystem. You will only be able to see the schema through the Schema API (if enabled) or through the Solr Admin UI's Cloud Screens.

Whichever name of the file is being used in your installation, the structure of the file is not changed. However, the way you interact with the file will change. If you are using the managed schema, it is expected that you only interact with the file with the Schema API, and never make manual edits. If you do not use the managed schema, you will only be able to make manual edits to the file, the Schema API will not support any modifications.

Note that if you are not using the Schema API yet you do use SolrCloud, you will need to interact with `schema.xml` through ZooKeeper using upconfig and downconfig commands to make a local copy and upload your changes. The options for doing this are described in Solr Start Script Reference and Using ZooKeeper to Manage Configuration Files.

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`. Each field type is defined between `fieldType` elements. They can optionally be grouped within a `types` element. 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"
/>
    <!-- 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"
/>
    <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

These are properties that can be specified either on the field types, or on individual fields to override the values provided by the field types. The default values for each property depend on the underlying `FieldType` class, which in turn may depend on the `version` attribute of the `<schema/>`. The table below includes the default value for most `FieldType` implementations provided by Solr, assuming a `schema.xml` that declares `version="1.6"`.

Property	Description	Values	Implicit Default
indexed	If true, the value of the field can be used in queries to retrieve matching documents.	true or false	true
stored	If true, the actual value of the field can be retrieved by queries.	true or false	true
docValues	If true, the value of the field will be put in a column-oriented DocValues structure.	true or false	false
sortMissingFirst sortMissingLast	Control the placement of documents when a sort field is not present.	true or false	false
multiValued	If true, indicates that a single document might contain multiple values for this field type.	true or false	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 field types that are not text fields.**	true or false	*
omitPositions	Similar to `omitTermFreqAndPositions` but preserves term frequency information.	true or false	*
termVectors termPositions termOffsets termPayloads	These options instruct Solr to maintain full term vectors for each document, optionally including position, offset and payload 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	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	false
useDocValuesAsStored	If the field has docValues enabled, setting this to true would allow the field to be returned as if it were a stored field (even if it has `stored=false`) when matching "`*`" in an fl parameter.	true or false	true

Field Type Similarity

A field type may optionally specify a `<similarity/>` that will be used when scoring documents that refer to fields with this type, as long as the "global" similarity for the collection allows it. By default, any field type which does not define a similarity, uses `BM25Similarity`. For more details, and examples of configuring both global & per-type Similarities, please see Other Schema Elements.

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
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.

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.
DateRangeField	Supports indexing date ranges, to include point in time date instances as well (single-millisecond durations). See the section Working with Dates for more detail on using this field type. Consider using this field type even if it's just for date instances, particularly when the queries typically fall on UTC year/month/day/hour, etc., boundaries.
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.
ICUCollationField	Supports Unicode collation for sorting and range queries. See the section Unicode Collation.
LatLonType	Spatial Search: a latitude/longitude coordinate pair. The latitude is specified first in the pair.
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. Configuration and usage of PreAnalyzedField is documented on the Working with External Files and Processes page.
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.
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). Strings are intended for small fields and are *not* tokenized or analyzed in any way. They have a hard limit of slightly less than 32K.
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.

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
- 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

You can configure Solr to download exchange rates from OpenExchangeRates.Org, with updates rates between USD and 170 currencies hourly. These rates are symmetrical only.

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

```
<fieldType name="currency" class="solr.CurrencyField" precisionStep="8"
           providerClass="solr.OpenExchangeRatesOrgProvider"
           refreshInterval="60"

ratesFileLocation="http://www.openexchangerates.org/api/latest.json?app_id=yourPersonalAppIdKey"/>
```

The `refreshInterval` is minutes, so the above example will download the newest rates every 60 minutes. The refresh interval may be increased, but not decreased.

Working with Dates

Date Formatting

Solr's date fields (`TrieDateField` and `DateRangeField`) 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 – a restricted subset of ISO-8601. For those familiar with Java 8, Solr uses DateTimeFormatter.ISO_INSTANT for formatting, and parsing too with "leniency".

`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 example values with sub-seconds:

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

There must be a leading '`-`' for dates prior to year 0000, and Solr will format dates with a leading '`+`' for years after 9999. Year 0000 is considered year 1 BC; there is no such thing as year 0 AD or BC.

Query escaping may be required
As you can see, the date format includes colon characters separating the hours, minutes, and seconds. Because the colon is a special character to Solr's most common query parsers, escaping is sometimes required, depending on exactly what you are trying to do.

This is normally an invalid query:
```
datefield:1972-05-20T17:33:18.772Z
```

These are valid queries:
```
datefield:1972-05-20T17\:33\:18.772Z
datefield:"1972-05-20T17:33:18.772Z"
datefield:[1972-05-20T17:33:18.772 TO *]
```

Date Range Formatting

Solr's `DateRangeField` supports the same point in time date syntax described above (with *date math* described below) and more to express date ranges. One class of examples is truncated dates, which represent the entire date span to the precision indicated. The other class uses the range syntax (`[ TO ]`). Here are some examples:

- `2000-11` – The entire month of November, 2000.
- `2000-11T13` – Likewise but for the 13th hour of the day (1pm-2pm).
- `-0009` – The year 10 BC. A 0 in the year position is 0 AD, and is also considered 1 BC.

- `[2000-11-01 TO 2014-12-01]` – The specified date range at a day resolution.
- `[2014 TO 2014-12-01]` – From the start of 2014 till the end of the first day of December.
- `[* TO 2014-12-01]` – From the earliest representable time thru till the end of 2014-12-01.

Limitations: The range syntax doesn't support embedded date math. If you specify a date instance supported by TrieDateField with date math truncating it, like `NOW/DAY`, you still get the first millisecond of that day, not the entire day's range. Exclusive ranges (using { & }) work in queries but not for indexing ranges.

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/my_collection/select?q=*:*&facet.range=my_date_field&face
t=true&facet.range.start=NOW/MONTH&facet.range.end=NOW/MONTH%2B1MONTH&facet.range.ga
p=%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/my_collection/select?q=*:*&facet.range=my_date_field&face
t=true&facet.range.start=NOW/MONTH&facet.range.end=NOW/MONTH%2B1MONTH&facet.range.ga
p=%2B1DAY&TZ=America/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>
...
```

More DateRangeField Details

`DateRangeField` is almost a drop-in replacement for places where `TrieDateField` is used. The only difference is that Solr's XML or SolrJ response formats will expose the stored data as a String instead of a Date. The underlying index data for this field will be a bit larger. Queries that align to units of time a second on up should be faster than TrieDateField, especially if it's in UTC. But the main point of DateRangeField as it's name suggests is to allow indexing date ranges. To do that, simply supply strings in the format shown above. It also supports specifying 3 different relational predicates between the indexed data, and the query range: `Intersects` (default), `Contains`, `Within`. You can specify the predicate by querying using the `op` local-params parameter like so:

```
fq={!field f=dateRange op=Contains}[2013 TO 2018]
```

In that example, it would find documents with indexed ranges that *contain* (or equals) the range 2013 thru 2018. Multi-valued overlapping indexed ranges in a document are effectively coalesced.

For a DateRangeField example use-case and possibly other information, see Solr's community wiki.

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="priority"/>
<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
  - Format of the External File
  - Reloading an External File
- The PreAnalyzedField Type
  - JsonPreAnalyzedParser
  - SimplePreAnalyzedParser

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

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"/>
```

The `PreAnalyzedField` 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`.

By default, the query-time analyzer for fields of this type will be the same as the index-time analyzer, which expects serialized pre-analyzed text. You must add a query type analyzer to your fieldType in order to perform analysis on non-pre-analyzed queries. In the example below, the index-time analyzer expects the default JSON serialization format, and the query-time analyzer will employ StandardTokenizer/LowerCaseFilter:

```
<fieldType name="pre_with_query_analyzer" class="solr.PreAnalyzedField">
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

JsonPreAnalyzedParser

This is the default serialization format used by PreAnalyzedField type. It uses a top-level JSON map with the following keys:

Key	Description	Required?
v	Version key. Currently the supported version is `1`.	required
`str`	Stored string value of a field. You can use at most one of `str` or `bin`.	optional

`bin`	Stored binary value of a field. The binary value has to be Base64 encoded.	optional
`tokens`	serialized token stream. This is a JSON list.	optional

Any other top-level key is silently ignored.

Token stream serialization

The token stream is expressed as a JSON list of JSON maps. The map for each token consists of the following keys and values:

Key	Description	Lucene Attribute	Value	Required?
`t`	token	CharTermAttribute	UTF-8 string representing the current token	required
`s`	start offset	OffsetAttribute	Non-negative integer	optional
`e`	end offset	OffsetAttribute	Non-negative integer	optional
`i`	position increment	PositionIncrementAttribute	Non-negative integer - default is `1`	optional
`p`	payload	PayloadAttribute	Base64 encoded payload	optional
`y`	lexical type	TypeAttribute	UTF-8 string	optional
`f`	flags	FlagsAttribute	String representing an integer value in hexadecimal format	optional

Any other key is silently ignored.

Example

```
{
  "v":"1",
  "str":"test ó",
  "tokens": [
    {"t":"one","s":123,"e":128,"i":22,"p":"DQ4KDQsODg8=","y":"word"},
    {"t":"two","s":5,"e":8,"i":1,"y":"word"},
    {"t":"three","s":20,"e":22,"i":1,"y":"foobar"}
  ]
}
```

SimplePreAnalyzedParser

The fully qualified class name to use when specifying this format via the `parserImpl` configuration parameter is `org.apache.solr.schema.SimplePreAnalyzedParser`.

Syntax

The serialization format supported by this parser is as follows:

Serialization format

```
content ::= version (stored)? tokens
version ::= digit+ " "
; stored field value - any "=" inside must be escaped!
stored ::= "=" text "="
tokens ::= (token ((" ") + token)*)*
token ::= text ("," attrib)*
attrib ::= name '=' value
name ::= text
value ::= text
```

Special characters in "text" values can be escaped using the escape character `\` . The following escape sequences are recognized:

Escape Sequence	Description
"\ "	literal space character
"\,"	literal , character
"\="	literal = character
"\\"	literal \ character
"\n"	newline
"\r"	carriage return
"\t"	horizontal tab

Please note that Unicode sequences (e.g. `\u0001`) are not supported.

Supported attribute names

The following token attributes are supported, and identified with short symbolic names:

Name	Description	Lucene attribute	Value format
i	position increment	PositionIncrementAttribute	integer
s	start offset	OffsetAttribute	integer
e	end offset	OffsetAttribute	integer
y	lexical type	TypeAttribute	string
f	flags	FlagsAttribute	hexadecimal integer
p	payload	PayloadAttribute	bytes in hexadecimal format; whitespace is ignored

Token positions are tracked and implicitly added to the token stream - the start and end offsets consider only the term text and whitespace, and exclude the space taken by token attributes.

Example token streams

```
1 one two three
```

- version: 1
- stored: null
- token: (term=one,startOffset=0,endOffset=3)
- token: (term=two,startOffset=4,endOffset=7)
- token: (term=three,startOffset=8,endOffset=13)

```
1 one  two    three
```

- version: 1
- stored: null
- token: (term=one,startOffset=0,endOffset=3)
- token: (term=two,startOffset=5,endOffset=8)
- token: (term=three,startOffset=11,endOffset=16)

```
1 one,s=123,e=128,i=22 two three,s=20,e=22
```

- version: 1
- stored: null
- token: (term=one,positionIncrement=22,startOffset=123,endOffset=128)
- token: (term=two,positionIncrement=1,startOffset=5,endOffset=8)
- token: (term=three,positionIncrement=1,startOffset=20,endOffset=22)

```
1 \ one\ \,,i=22,a=\, two\=

\n,\ =\ \
```

- version: 1
- stored: null
- token: (term= one ,,positionIncrement=22,startOffset=0,endOffset=6)
- token: (term=two=

  ,positionIncrement=1,startOffset=7,endOffset=15)
- token: (term=\,positionIncrement=1,startOffset=17,endOffset=18)

Note that unknown attributes and their values are ignored, so in this example, the "a" attribute on the first token and the " " (escaped space) attribute on the second token are ignored, along with their values, because they are not among the supported attribute names.

```
1 ,i=22 ,i=33,s=2,e=20 ,
```

- version: 1
- stored: null
- token: (term=,positionIncrement=22,startOffset=0,endOffset=0)
- token: (term=,positionIncrement=33,startOffset=2,endOffset=20)
- token: (term=,positionIncrement=1,startOffset=2,endOffset=2)

```
1 =This is the stored part with \=
\n \t escapes.=one two three
```

- version: 1
- stored: "`This is the stored part with =

\t escapes.`"
- token: (term=`one`,startOffset=0,endOffset=3)
- token: (term=`two`,startOffset=4,endOffset=7)
- token: (term=`three`,startOffset=8,endOffset=13)

Note that the "`\t`" in the above stored value is not literal; it's shown that way to visually indicate the actual tab char that is in the stored value.

```
1 ==
```

- version: 1
- stored: ""
- (no tokens)

```
1 =this is a test.=
```

- version: 1
- stored: "this is a test."
- (no tokens)

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 many of the same properties as field types. Properties from the table below which are specified on an individual field will override any explicit value for that property specified on the the `<fieldType/>` of the field, or any implicit default property value provided by the underlying `FieldType` implementation. The table below is reproduced from Field Type Definitions and Properties, which has more details:

Property	Description	Values	Implicit Default
indexed	If true, the value of the field can be used in queries to retrieve matching documents.	true or false	true
stored	If true, the actual value of the field can be retrieved by queries.	true or false	true
docValues	If true, the value of the field will be put in a column-oriented DocValues structure.	true or false	false
sortMissingFirst sortMissingLast	Control the placement of documents when a sort field is not present.	true or false	false
multiValued	If true, indicates that a single document might contain multiple values for this field type.	true or false	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 field types that are not text fields.**	true or false	*
omitPositions	Similar to `omitTermFreqAndPositions` but preserves term frequency information.	true or false	*
termVectors termPositions termOffsets termPayloads	These options instruct Solr to maintain full term vectors for each document, optionally including position, offset and payload 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	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	false

useDocValuesAsStored	If the field has docValues enabled, setting this to true would allow the field to be returned as if it were a stored field (even if it has `stored=false`) when matching "*" in an fl parameter.	true or false	true

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" />
```

In this example, we want Solr to copy the `cat` field to a field named `text`. Fields are copied before analysis is done, meaning you can have two fields with identical original content, but which use different analysis chains and are stored in the index differently.

In the example above, 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 from multiple `copyField` directives).

A common usage for this functionality is to create a single "search" field that will serve as the default query field when users or clients do not specify a field to query. For example, `title`, `author`, `keywords`, and `body` may all be fields that should be searched by default, with copy field rules for each field to copy to a `catchall` field (for example, it could be named anything). Later you can set a rule in `solrconfig.xml` to search the `catchall` field by default. One caveat to this is your index will grow when using copy fields. However, whether this becomes problematic for you and the final size will depend on the number of fields being copied, the number of destination fields being copied to, the analysis in use, and the available disk space.

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.

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`.

Like regular fields, dynamic fields 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>
```

Schema defaults and `copyFields` cannot be used to populate the `uniqueKey` field. You can 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 & Query Operator

Although they have been deprecated for quite some time, Solr still has support for Schema based configuration of a `<defaultSearchField/>` (which is superseded by the `df parameter`) and `<solrQueryParser defaultOperator="OR"/>` (which is superseded by the `q.op` parameter.

If you have these options specified in your Schema, you are strongly encouraged to replace them with request parameters (or request parameter defaults) as support for them may be removed from future Solr release.

Similarity

Similarity is a Lucene class used to score a document in searching.

Each collection has one "global" Similarity, and by default Solr uses an implicit `SchemaSimilarityFactory` which allows individual field types to be configured with a "per-type" specific Similarity and implicitly uses `BM25Similarity` for any field type which does not have an explicit Similarity.

This default behavior can be overridden by declaring a top level `<similarity/>` element in your `schema.xml`, outside of any single field type. This similarity declaration can either refer directly to the name of a class with a no-argument constructor, such as in this example showing `BM25Similarity`:

```
<similarity class="solr.BM25Similarity"/>
```

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>
```

In most cases, specifying global level similarity like this will cause an error if your `schema.xml` also includes field type specific `<similarity/>` declarations. One key exception to this is that you may explicitly declare a `SchemaSimilarityFactory` and specify what that default behavior will be for all field types that do not declare an explicit Similarity using the name of field type (specified by `defaultSimFromFieldType`) that *is* configured with a specific similarity:

```
<similarity class="solr.SchemaSimilarityFactory">
  <str name="defaultSimFromFieldType">text_dfr</str>
<similarity>
<fieldType name="text_dfr" class="solr.TextField">
  <analyzer ... />
  <similarity class="solr.DFRSimilarityFactory">
    <str name="basicModel">I(F)</str>
    <str name="afterEffect">B</str>
    <str name="normalization">H3</str>
    <float name="mu">900</float>
  </similarity>
</fieldType>
<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>
<fieldType name="text_other">
  <analyzer ... />
</fieldType>
```

In the example above `IBSimilarityFactory` (using the Information-Based model) will be used for any fields of type `text_ib`, while `DFRSimilarityFactory` (divergence from random) will be used for any fields of type `t`

`ext_dfr`, as well as any fields using a type that does not explicitly specify a `<similarity/>`.

If `SchemaSimilarityFactory` is explicitly declared with out configuring a `defaultSimFromFieldType`, then `BM25Similarity` is implicitly used as the default.

In addition to the various factories mentioned on this page, there are several other similarity implementations that can be used such as the `SweetSpotSimilarityFactory`, `ClassicSimilarityFactory`, etc.... For details, see the Solr Javadocs for the similarity factories.

Schema API

The Schema API provides read and write access to the Solr schema for each collection (or core, when using standalone Solr). Read access to all schema elements is supported. Fields, dynamic fields, field types and copyField rules may be added, removed or replaced. Future Solr releases will extend write access to allow more schema elements to be modified.



Re-index after schema modifications!
If you modify your schema, you will likely need to re-index all documents. If you do not, you may lose access to documents, or not be able to interpret them properly, e.g. after replacing a field type.

Modifying your schema will never modify any documents that are already indexed. Again, you must re-index documents in order to apply schema changes to them.

To enable schema modification with this API, the schema will need to be managed and mutable. See the section Schema Factory 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.

When modifying the schema with the API, a core reload will automatically occur in order for the changes to be available immediately for documents indexed thereafter. Previously indexed documents will **not** be automatically handled - they **must** be re-indexed if they used schema elements that you changed.

The base address for the API is `http://<host>:<port>/solr/<collection_name>`. If for example you run Solr's "`cloud`" example (via the `bin/solr` command shown below), which creates a "`gettingstarted`" collection, then the base URL (as in all the sample URLs in this section) would be: `http://localhost:8983/solr/gettingstarted` .

```
bin/solr -e cloud -noprompt
```


- API Entry Points
- Modify the Schema
  - Add a New Field
  - Delete a Field
  - Replace a Field
  - Add a Dynamic Field Rule
  - Delete a Dynamic Field Rule
  - Replace a Dynamic Field Rule
  - Add a New Field Type
  - Delete a Field Type
  - Replace a Field Type
  - Add a New Copy Field Rule
  - Delete a Copy Field Rule
  - Multiple Commands in a Single POST

    - Schema Changes among Replicas
- Retrieve Schema Information
    - Retrieve the Entire Schema
    - List Fields
    - List Dynamic Fields
    - List Field Types
    - List Copy Fields
    - Show Schema Name
    - Show the Schema Version
    - List UniqueKey
    - Show Global Similarity
    - Get the Default Query Operator
- Manage Resource Data

API Entry Points

`/schema`: retrieve the schema, or modify the schema to add, remove, or replace fields, dynamic fields, copy fields, or field types
`/schema/fields`: retrieve information about all defined fields or a specific named field
`/schema/dynamicfields`: retrieve information about all dynamic field rules or a specific named dynamic rule
`/schema/fieldtypes`: retrieve information about all field types or a specific field type
`/schema/copyfields`: retrieve information about copy fields
`/schema/name`: retrieve the schema name
`/schema/version`: retrieve the schema version
`/schema/uniquekey`: retrieve the defined uniqueKey
`/schema/similarity`: retrieve the global similarity definition
`/schema/solrqueryparser/defaultoperator`: retrieve the default operator

Modify the Schema

`POST /collection/schema`

To add, remove or replace fields, dynamic field rules, copy field rules, or new field types, you can send a POST request to the `/collection/schema/` endpoint with a sequence of commands to perform the requested actions. The following commands are supported:

- `add-field`: add a new field with parameters you provide.
- `delete-field`: delete a field.
- `replace-field`: replace an existing field with one that is differently configured.

- `add-dynamic-field`: add a new dynamic field rule with parameters you provide.
- `delete-dynamic-field`: delete a dynamic field rule.
- `replace-dynamic-field`: replace an existing dynamic field rule with one that is differently configured.

- `add-field-type`: add a new field type with parameters you provide.
- `delete-field-type`: delete a field type.
- `replace-field-type`: replace an existing field type with one that is differently configured.

- `add-copy-field`: add a new copy field rule.
- `delete-copy-field`: delete a copy field rule.

These commands can be issued in separate POST requests or in the same POST request. Commands are executed in the order in which they are specified.

In each case, the response will include the status and the time to process the request, but will not include the entire schema.

When modifying the schema with the API, a core reload will automatically occur in order for the changes to be available immediately for documents indexed thereafter. Previously indexed documents will **not** be automatically handled - they **must** be re-indexed if they used schema elements that you changed.

Add a New Field

The `add-field` command adds a new field definition to your schema. If a field with the same name exists an error is thrown.

All of the properties available when defining a field with manual `schema.xml` edits can be passed via the API. These request attributes are described in detail in the section Defining Fields.

For example, to define a new stored field named "sell-by", of type "tdate", you would POST the following request:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
     "name":"sell-by",
     "type":"tdate",
     "stored":true }
}' http://localhost:8983/solr/gettingstarted/schema
```

Delete a Field

The `delete-field` command removes a field definition from your schema. If the field does not exist in the schema, or if the field is the source or destination of a copy field rule, an error is thrown.

For example, to delete a field named "sell-by", you would POST the following request:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "delete-field" : { "name":"sell-by" }
}' http://localhost:8983/solr/gettingstarted/schema
```

Replace a Field

The `replace-field` command replaces a field's definition. Note that you must supply the full definition for a field - this command will **not** partially modify a field's definition. If the field does not exist in the schema an error is thrown.

All of the properties available when defining a field with manual `schema.xml` edits can be passed via the API. These request attributes are described in detail in the section Defining Fields.

For example, to replace the definition of an existing field "sell-by", to make it be of type "date" and to not be stored, you would POST the following request:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "replace-field":{
     "name":"sell-by",
     "type":"date",
     "stored":false }
}' http://localhost:8983/solr/gettingstarted/schema
```

Add a Dynamic Field Rule

The `add-dynamic-field` command adds a new dynamic field rule to your schema.

All of the properties available when editing `schema.xml` can be passed with the POST request. The section Dynamic Fields has details on all of the attributes that can be defined for a dynamic field rule.

For example, to create a new dynamic field rule where all incoming fields ending with "_s" would be stored and have field type "string", you can POST a request like this:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-dynamic-field":{
     "name":"*_s",
     "type":"string",
     "stored":true }
}' http://localhost:8983/solr/gettingstarted/schema
```

Delete a Dynamic Field Rule

The `delete-dynamic-field` command deletes a dynamic field rule from your schema. If the dynamic field rule does not exist in the schema, or if the schema contains a copy field rule with a target or destination that matches only this dynamic field rule, an error is thrown.

For example, to delete a dynamic field rule matching "*_s", you can POST a request like this:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "delete-dynamic-field":{ "name":"*_s" }
}' http://localhost:8983/solr/gettingstarted/schema
```

Replace a Dynamic Field Rule

The `replace-dynamic-field` command replaces a dynamic field rule in your schema. Note that you must supply the full definition for a dynamic field rule - this command will **not** partially modify a dynamic field rule's definition. If the dynamic field rule does not exist in the schema an error is thrown.

All of the properties available when editing `schema.xml` can be passed with the POST request. The section Dynamic Fields has details on all of the attributes that can be defined for a dynamic field rule.

For example, to replace the definition of the "*_s" dynamic field rule with one where the field type is "text_general" and it's not stored, you can POST a request like this:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "replace-dynamic-field":{
     "name":"*_s",
     "type":"text_general",
     "stored":false }
}' http://localhost:8983/solr/gettingstarted/schema
```

Add a New Field Type

The `add-field-type` command adds a new field type to your schema.

All of the field type properties available when editing `schema.xml` by hand are available for use in a POST

request. The structure of the command is a json mapping of the standard field type definition, including the name, class, index and query analyzer definitions, etc. Details of all of the available options are described in the section Solr Field Types.

For example, to create a new field type named "myNewTxtField", you can POST a request as follows:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field-type" : {
     "name":"myNewTxtField",
     "class":"solr.TextField",
     "positionIncrementGap":"100",
     "analyzer" : {
        "charFilters":[{
           "class":"solr.PatternReplaceCharFilterFactory",
           "replacement":"$1$1",
           "pattern":"([a-zA-Z])\\\\1+" }],
        "tokenizer":{
           "class":"solr.WhitespaceTokenizerFactory" },
        "filters":[{
           "class":"solr.WordDelimiterFilterFactory",
           "preserveOriginal":"0" }]}}
}' http://localhost:8983/solr/gettingstarted/schema
```

Note in this example that we have only defined a single analyzer section that will apply to index analysis and query analysis. If we wanted to define separate analysis, we would replace the `analyzer` section in the above example with separate sections for `indexAnalyzer` and `queryAnalyzer`. As in this example:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field-type":{
     "name":"myNewTextField",
     "class":"solr.TextField",
     "indexAnalyzer":{
        "tokenizer":{
           "class":"solr.PathHierarchyTokenizerFactory",
           "delimiter":"/" }},
     "queryAnalyzer":{
        "tokenizer":{
           "class":"solr.KeywordTokenizerFactory" }}}
}' http://localhost:8983/solr/gettingstarted/schema
```

Delete a Field Type

The `delete-field-type` command removes a field type from your schema. If the field type does not exist in the schema, or if any field or dynamic field rule in the schema uses the field type, an error is thrown.

For example, to delete the field type named "myNewTxtField", you can make a POST request as follows:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "delete-field-type":{ "name":"myNewTxtField" }
}' http://localhost:8983/solr/gettingstarted/schema
```

Replace a Field Type

The `replace-field-type` command replaces a field type in your schema. Note that you must supply the full definition for a field type - this command will **not** partially modify a field type's definition. If the field type does not exist in the schema an error is thrown.

All of the field type properties available when editing `schema.xml` by hand are available for use in a POST request. The structure of the command is a json mapping of the standard field type definition, including the name, class, index and query analyzer definitions, etc. Details of all of the available options are described in the section Solr Field Types.

For example, to replace the definition of a field type named "myNewTxtField", you can make a POST request as follows:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "replace-field-type":{
     "name":"myNewTxtField",
     "class":"solr.TextField",
     "positionIncrementGap":"100",
     "analyzer":{
        "tokenizer":{
           "class":"solr.StandardTokenizerFactory" }}}
}' http://localhost:8983/solr/gettingstarted/schema
```

Add a New Copy Field Rule

The `add-copy-field` command adds a new copy field rule to your schema.

The attributes supported by the command are the same as when creating copy field rules by manually editing the `schema.xml`, as below:

Name	Required	Description
source	Yes	The source field.
dest	Yes	A field or an array of fields to which the source field will be copied.
maxChars	No	The upper limit for the number of characters to be copied. The section Copying Fields has more details.

For example, to define a rule to copy the field "shelf" to the "location" and "catchall" fields, you would POST the following request:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-copy-field":{
     "source":"shelf",
     "dest":[ "location", "catchall" ]}
}' http://localhost:8983/solr/gettingstarted/schema
```

Delete a Copy Field Rule

The `delete-copy-field` command deletes a copy field rule from your schema. If the copy field rule does not exist in the schema an error is thrown.

The `source` and `dest` attributes are required by this command.

For example, to delete a rule to copy the field "shelf" to the "location" field, you would POST the following request:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "delete-copy-field":{ "source":"shelf", "dest":"location" }
}' http://localhost:8983/solr/gettingstarted/schema
```

Multiple Commands in a Single POST

It is possible to perform one or more add requests in a single command. The API is transactional and all commands in a single call either succeed or fail together.

The commands are executed in the order in which they are specified. This means that if you want to create a new field type and in the same request use the field type on a new field, the section of the request that creates the field type must come before the section that creates the new field. Similarly, since a field must exist for it to be used in a copy field rule, a request to add a field must come before a request for the field to be used as either the source or the destination for a copy field rule.

The syntax for making multiple requests supports several approaches. First, the commands can simply be made serially, as in this request to create a new field type and then a field that uses that type:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field-type":{
     "name":"myNewTxtField",
     "class":"solr.TextField",
     "positionIncrementGap":"100",
     "analyzer":{
        "charFilters":[{
           "class":"solr.PatternReplaceCharFilterFactory",
           "replacement":"$1$1",
           "pattern":"([a-zA-Z])\\\\1+" }],
        "tokenizer":{
           "class":"solr.WhitespaceTokenizerFactory" },
        "filters":[{
           "class":"solr.WordDelimiterFilterFactory",
           "preserveOriginal":"0" }]}},
   "add-field" : {
      "name":"sell-by",
      "type":"myNewTxtField",
      "stored":true }
}' http://localhost:8983/solr/gettingstarted/schema
```

Or, the same command can be repeated, as in this example:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
     "name":"shelf",
     "type":"myNewTxtField",
     "stored":true },
  "add-field":{
     "name":"location",
     "type":"myNewTxtField",
     "stored":true },
  "add-copy-field":{
     "source":"shelf",
      "dest":[ "location", "catchall" ]}
}' http://localhost:8983/solr/gettingstarted/schema
```

Finally, repeated commands can be sent as an array:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":[
     { "name":"shelf",
       "type":"myNewTxtField",
       "stored":true },
     { "name":"location",
       "type":"myNewTxtField",
       "stored":true }]
}' http://localhost:8983/solr/gettingstarted/schema
```

Schema Changes among Replicas

When running in SolrCloud mode, changes made to the schema on one node will propagate to all replicas in the collection. You can pass the **updateTimeoutSecs** parameter with your request to set the number of seconds to wait until all replicas confirm they applied the schema updates. This helps your client application be more robust in that you can be sure that all replicas have a given schema change within a defined amount of time. If agreement is not reached by all replicas in the specified time, then the request fails and the error message will include information about which replicas had trouble. In most cases, the only option is to re-try the change after waiting a brief amount of time. If the problem persists, then you'll likely need to investigate the server logs on the replicas that had trouble applying the changes. If you do not supply an **updateTimeoutSecs** parameter, the default behavior is for the receiving node to return immediately after persisting the updates to ZooKeeper. All other replicas will apply the updates asynchronously. Consequently, without supplying a timeout, your client application cannot be sure that all replicas have applied the changes.

Retrieve Schema Information

The following endpoints allow you to read how your schema has been defined. You can GET the entire schema, or only portions of it as needed.

To modify the schema, see the previous section Modify the Schema.

Retrieve the Entire Schema

`GET /collection/schema`

INPUT

Path Parameters

Key	Description
collection	The collection (or core) name.

Query Parameters

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

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, in the format requested (JSON or XML). The schema name and version are also included.

EXAMPLES

Get the entire schema in JSON.

```
curl http://localhost:8983/solr/gettingstarted/schema?wt=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"}]}}
```

Get the entire schema in XML.

```
curl http://localhost:8983/solr/gettingstarted/schema?wt=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>
```

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

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

```
<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`

`GET /collection/schema/fields/fieldname`

INPUT

Path Parameters

Key	Description
collection	The collection (or core) name.
fieldname	The specific fieldname (if limiting request to a single field).

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.
fl	string	No	(all fields)	Comma- or space-separated list of one or more fields to return. If not specified, all fields will be returned by default.
includeDynamic	boolean	No	false	If **true**, and if the **fl** query parameter is specified or the **fieldname** path parameter is used, matching dynamic fields are included in the response and identified with the **dynamicBase** property. If neither the **fl** query parameter nor the **fieldname** path parameter is specified, the **includeDynamic** query parameter is ignored. If **false**, matching dynamic fields will not be returned.

showDefaults	boolean	No	false	If **true**, all default field properties from each field's field type will be included in the response (e.g. **tokenized** for **solr.TextField**). If **false**, only explicitly specified field properties will be included.

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

Get a list of all fields.

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

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 Dynamic Fields

`GET /collection/schema/dynamicfields`

`GET /collection/schema/dynamicfields/name`

INPUT

Path Parameters

Key	Description
collection	The collection (or core) name.
name	The name of the dynamic field rule (if limiting request to a single 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, xml** . If not specified, JSON will be returned by default.
showDefaults	boolean	No	false	If **true**, all default field properties from each dynamic field's field type will be included in the response (e.g. **tokenized** for **solr.TextField**). If **false**, only explicitly specified field properties will be included.

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

Get a list of all dynamic field declarations:

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

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 Field Types

`GET /collection/schema/fieldtypes`

`GET /collection/schema/fieldtypes/name`

INPUT

Path Parameters

Key	Description
collection	The collection (or core) name.
name	The name of the field type (if limiting request to a single 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.
showDefaults	boolean	No	false	If **true**, all default field properties from each field type will be included in the response (e.g. **tokenized** for **solr.TextField**). If **false**, only explicitly specified field properties will be included.

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

Get a list of all field types.

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

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 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.
source.fl	string	No	(all source fields)	Comma- or space-separated list of one or more copyField source fields to include in the response - copyField directives with all other source fields will be excluded from the response. If not specified, all copyField-s will be included in the response.
dest.fl	string	No	(all dest fields)	Comma- or space-separated list of one or more copyField dest fields to include in the response - copyField directives with all other dest fields will be excluded. If not specified, all copyField-s will be included in the response.

OUTPUT

Output Content

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

EXAMPLES

Get a list of all copyfields.

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

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

Get the schema name.

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

```
{
  "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

Get the schema version

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

```
{
  "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

List the uniqueKey.

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

```
{
  "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

Get the similarity implementation.

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

```
{
  "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

Get the default operator.

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

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

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 secti on for more information and examples.

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>
  <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`. The `uniqueKey` must always be defined. In older Solr versions you would find `defaultSearchField` and `solrQueryParser` tags as well, but although these still work they are deprecated and discouraged, see Other Schema Elements.

Types and fields are optional tags
Note that the `types` and `fields` sections are optional, meaning you are free to mix `field`, `dynamicF ield`, `copyField` and `fieldType` definitions on the top level. This allows for a more logical grouping of related tags in your schema.

Choosing Appropriate Numeric Types

For general numeric needs, use `TrieIntField`, `TrieLongField`, `TrieFloatField`, and `TrieDoubleFiel d` 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 schema.xml file for the "`techproducts`" example included with Solr, `copyField` declarations are 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 `StrField`:

```
Schildt, Herbert
```

Related Topics

- SchemaXML

DocValues

DocValues are a way of recording field values internally that is more efficient for some purposes, such as sorting and faceting, than traditional indexing.

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.

Enabling 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 (or field type) definition, as in this example from the `schema.xml` of Solr's `sample_techproducts_configs` config set:

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



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:

- `StrField` and `UUIDField`.
  - 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*` numeric fields, date fields and `EnumField`.
  - 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.

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

There are two implications of multi-valued DocValues being stored as SORTED_SET types that should be kept in mind when combined with /export (and, by extension, Streaming Expression-based functionality):

1. Values are returned in sorted order rather than the original input order.
2. If multiple, identical entries are in the field in a *single* document, only one will be returned for that document.

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.

Using DocValues

Sorting, Faceting & Functions

If `docValues="true"` for a field, then DocValues will automatically be used any time the field is used for sorting, faceting or Function Queries.

Retrieving DocValues During Search

Field values retrieved during search queries are typically returned from stored values. However, non-stored docValues fields will be also returned along with other stored fields when all fields (or pattern matching globs) are specified to be returned (e.g. "`fl=*`") for search queries depending on the effective value of the `useDocValuesAsStored` parameter for each field. For schema versions >= 1.6, the implicit default is `useDocValuesAsStored="true"`. See Field Type Definitions and Properties & Defining Fields for more details.

When `useDocValuesAsStored="false"`, non-stored DocValues fields can still be explicitly requested by name in the fl param, but will not match glob patterns (`"*"`).

Note that returning DocValues along with "regular" stored fields at query time has performance implications that stored fields may not because DocValues are column-oriented and may therefore incur additional cost to retrieve for each returned document. Also note that while returning non-stored fields from DocValues, the values of a multi-valued field are returned in sorted order (and not insertion order). If you require the multi-valued fields to be returned in the original insertion order, then make your multi-valued field as stored (such a change requires re-indexing).

In cases where the query is returning *only* docValues fields performance may improve since returning stored fields requires disk reads and decompression whereas returning docValues fields in the fl list only requires memory access.

When retrieving fields from their docValues form, two important differences between regular stored fields and docValues fields must be understood:

1. Order is *not* preserved. For simply retrieving stored fields, the insertion order is the return order. For docValues, it is the *sorted* order.
2. Multiple identical entries are collapsed into a single value. Thus if I insert values 4, 5, 2, 4, 1, my return will be 1, 2, 4, 5.

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 controlled via `solrconfig.xml`, are:

1. Managed schema: Schema modifications are made at runtime through Solr APIs, which requires the use of `schemaFactory` that supports these changes - see Schema Factory Definition in SolrConfig for more details.
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 `data_driven_schema_configs` config set in the Solr distribution. To start an example instance of Solr using these configs, run the following command:

```
bin/solr start -e schemaless
```

This will launch a Solr server, and automatically create a collection (named "`gettingstarted`") that contains

only three fields in the initial schema: `id`, `_version_`, and `_text_`.

You can use the `/schema/fields` Schema API to confirm this: `curl http://localhost:8983/solr/gettingstarted/schema/fields` will output:

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



Because the `data_driven_schema_configs` config set includes a `copyField` directive that causes all content to be indexed in a predefined "catch-all" `_text_` field, to enable single-field search that includes all fields' content, the index will be larger than it would be without the `copyField`. When you nail down your schema, consider removing the `_text_` field and the corresponding `copyField` directive if you don't need it.

Configuring Schemaless Mode

As described above, there are three configuration elements that need to be in place to use Solr in schemaless mode. In the `data_driven_schema_configs` config set included with Solr these are already configured. 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 Schema Factory Definition in SolrConfig, Managed Schema support is enabled by default, unless your configuration specifies that `ClassicIndexSchemaFactory` should be used.

You can configure the `ManagedIndexSchemaFactory` (and control the resource file used, or disable future modifications) by adding an explicit `<schemaFactory/>` like the one below, please see Schema Factory Definition in SolrConfig for more details on the options available.

```
<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.FieldNameMutatingUpdateProcessorFactory">
    <str name="pattern">[^\w-\.]</str>
    <str name="replacement">_</str>
  </processor>
  <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">strings</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
- RemoveBlankFieldUpdateProcessorFactory
- FieldNameMutatingUpdateProcessorFactory
- ParseBooleanFieldUpdateProcessorFactory
- ParseLongFieldUpdateProcessorFactory
- ParseDoubleFieldUpdateProcessorFactory
- ParseDateFieldUpdateProcessorFactory
- AddSchemaFieldsUpdateProcessorFactory

Make the UpdateRequestProcessorChain the Default for the UpdateRequestHandler

Once the UpdateRequestProcessorChain has been defined, you must instruct your UpdateRequestHandlers to use it when working with index updates (i.e., adding, removing, replacing documents). Here is an example using InitParams to set the defaults on all `/update` request handlers:

```
<initParams path="/update/**">
    <lst name="defaults">
      <str name="update.chain">add-unknown-fields-to-the-schema</str>
    </lst>
  </initParams>
```



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 `data_driven_schema_configs`), 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/gettingstarted/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/gettingstarted/schema/fields`):

```
{
  "responseHeader":{
    "status":0,
    "QTime":1},
  "fields":[{
      "name":"Album",
      "type":"strings"},      // Field value guessed as String -> strings fieldType
    {
      "name":"Artist",
      "type":"strings"},      // Field value guessed as String -> strings 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":"_text_",
...
    },
    {
      "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 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/gettingstarted/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 you 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 (in the same `conf/` directory as `solrconfig.xml`).

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.core.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`.



Field Values versus Indexed Terms
The output of an Analyzer affects the *terms* indexed in a given field (and the terms used when parsing queries against those fields) but it has no impact on the *stored* value for the fields. For example: an analyzer might split "Brown Cow" into two indexed terms "brown" and "cow", but the stored value will still be a single String: "Brown Cow"

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.

Analysis for Multi-Term Expansion

In some types of queries (ie: Prefix, Wildcard, Regex, etc...) the input provided by the user is not natural language intended for Analysis. Things like Synonyms or Stop word filtering do not work in a logical way in these types of Queries.

The analysis factories that *can* work in these types of queries (such as Lowercasing, or Normalizing Factories) are known as `MultiTermAwareComponents`. When Solr needs to perform analysis for a query that results in Multi-Term expansion, only the `MultiTermAwareComponents` used in the `query` analyzer are used, Factory that is not Multi-Term aware will be skipped.

For most use cases, this provides the best possible behavior, but if you wish for absolute control over the analysis performed on these types of queries, you may explicitly define a `multiterm` analyzer to use, such as in the following 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>
  <!-- No analysis at all when doing queries that involved Multi-Term expansion -->
  <analyzer type="multiterm">
    <tokenizer class="solr.KeywordTokenizerFactory" />
  </analyzer>
</fieldType>
```

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 `TokenizerFactory` API. This factory class will be called upon to create new tokenizer instances as needed. Objects created by the factory must derive from `Tokenizer`, 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 `PatternReplaceFilte`

`rFactory`) 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:

- Standard Tokenizer
- Classic Tokenizer
- Keyword Tokenizer
- Letter Tokenizer
- Lower Case Tokenizer
- N-Gram Tokenizer
- Edge N-Gram Tokenizer
- ICU Tokenizer
- Path Hierarchy Tokenizer
- Regular Expression Pattern Tokenizer
- UAX29 URL Email Tokenizer
- White Space Tokenizer

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 user tips 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>
```

To use this tokenizer, you must add additional .jars to Solr's classpath (as described in the section Lib Directives in SolrConfig). See the `solr/contrib/analysis-extras/README.txt` for information on which jars you need to add to your `SOLR_HOME/lib`.

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 and preserves as single tokens the following:
  - Internet domain names containing top-level domains validated against the white list in the IANA Root Zone Database when the tokenizer was generated
  - email addresses
  - `file://`, `http(s)://`, and `ftp://` URLs
  - IPv4 and IPv6 addresses

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 `<HIRAGAN`

A>.

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", "e", "mail", "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 tokens.

Factory class: `solr.WhitespaceTokenizerFactory`

Arguments: `rule` : Specifies how to define whitespace for the purpose of tokenization. Valid values:

- `java`: (Default) Uses Character.isWhitespace(int)
- `unicode`: Uses Unicode's WHITESPACE property

Example:

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

In: "To be, or what?"

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

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.

```
<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 user tips about Solr's filters, see http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters.

Filters discussed in this section:

- ASCII Folding Filter
- Beider-Morse Filter
- Classic Filter
- Common Grams Filter
- Collation Key Filter
- Daitch-Mokotoff Soundex Filter
- Double Metaphone Filter
- Edge N-Gram Filter
- English Minimal Stem Filter
- Fingerprint Filter
- Hunspell Stem Filter
- Hyphenated Words Filter
- ICU Folding Filter
- ICU Normalizer 2 Filter
- ICU Transform Filter
- Keep Word Filter
- KStem Filter
- Length Filter
- Lower Case Filter
- Managed Stop Filter
- Managed Synonym Filter
- N-Gram Filter
- Numeric Payload Token Filter
- Pattern Replace Filter
- Phonetic Filter
- Porter Stem Filter
- Remove Duplicates Token Filter
- Reversed Wildcard Filter
- Shingle Filter
- Snowball Porter Stemmer Filter
- Standard Filter
- Stop Filter
- Suggest Stop Filter
- Synonym Filter
- Token Offset Payload Filter
- Trim Filter
- Type As Payload Filter
- Type Token Filter
- Word Delimiter Filter

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.



BeiderMorseFilter changed its behavior in Solr 5.0 due to an update to version 3.04 of the BMPM algorithm. Older version of Solr implemented BMPM version 3.00 (see http://stevemorse.org/phoneticinfo.htm). Any index built using this filter with earlier versions of Solr will need to be rebuilt.

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.

Daitch-Mokotoff Soundex Filter

Implements the Daitch-Mokotoff Soundex algorithm, which allows identification of similar names, even if they are spelled differently. More information about how this works is available in the section on Phonetic Matching.

Factory class: `solr.DaitchMokotoffSoundexFilterFactory`

Arguments:

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

Example:

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DaitchMokotoffSoundexFilterFactory" inject="true"/>
</analyzer>
```

Double Metaphone Filter

This filter creates tokens using the `DoubleMetaphone` encoding algorithm from commons-codec. For more information, see the Phonetic Matching section.

Factory class: `solr.DoubleMetaphoneFilterFactory`

Arguments:

`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.

Example:

Default behavior for inject (true): keep the original token and add phonetic token(s) at the same position.

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

In: "four score and Kuczewski"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "Kuczewski"(4)

Out: "four"(1), "FR"(1), "score"(2), "SKR"(2), "and"(3), "ANT"(3), "Kuczewski"(4), "KSSK"(4), "KXFS"(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). Note that "Kuczewski" has two encodings, which are added at the same position.

Example:

Discard original token (`inject="false"`).

```
<analyzer>
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.DoubleMetaphoneFilterFactory" inject="false"/>
</analyzer>
```

In: "four score and Kuczewski"

Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "Kuczewski"(4)

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

Note that "Kuczewski" has two encodings, which are added at the same position.

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"

Fingerprint Filter

This filter outputs a single token which is a concatenation of the sorted and de-duplicated set of input tokens. This can be useful for clustering/linking use cases.

Factory class: `solr.FingerprintFilterFactory`

Arguments:

`separator` : The character used to separate tokens combined into the single output token. Defaults to " " (a space character).

`maxOutputTokenSize` : The maximum length of the summarized output token. If exceeded, no output token is emitted. Defaults to 1024.

Example:

```
<analyzer type="index">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.FingerprintFilterFactory" separator="_" />
</analyzer>
```

In: "the quick brown fox jumped over the lazy dog"

Tokenizer to Filter: "the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"

Out: "brown_dog_fox_jumped_lazy_over_quick_the"

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="nfkc_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 Word 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**.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

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.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

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_name/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_name/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. For more information, see the section on Phonetic Matching.

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" (v2.0), "ColognePhonetic", or "Nysiis".

`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"

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.ReversedWildcardFilterFactory`

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.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.4` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

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)

Suggest Stop Filter

Like Stop Filter, this filter discards, or *stops* analysis of, tokens that are on the given stop words list. Suggest Stop Filter differs from Stop Filter in that it will not remove the last token unless it is followed by a token separator. For example, a query "`find the`" would preserve the '`the`' since it was not followed by a space, punctuation etc., and mark it as a `KEYWORD` so that following filters will not change or remove it. By contrast, a query like "`find the popsicle`" would remove "`the`" as a stopword, since it's followed by a space. When using one of the analyzing suggesters, you would normally use the ordinary `StopFilterFactory` in your index analyzer and then SuggestStopFilter in your query analyzer.

Factory class: `solr.SuggestStopFilterFactory`

Arguments:

`words`: (optional; default: `StopAnalyzer#ENGLISH_STOP_WORDS_SET`) The name of a stopwords file to parse.

`format`: (optional; default: `wordset`) Defines how the words file will be parsed. If `words` is not specified, then `format` must not be specified. The valid values for the format option are:

- `wordset`: This is the default format, which supports one word per line (including any intra-word whitespace) and allows whole line comments begining with the "#" character. Blank lines are ignored.
- `snowball`: This format allows for multiple words specified on each line, and trailing comments may be specified using the vertical line ("|"). Blank lines are ignored.

`ignoreCase`: (optional; default: `false`) If `true`, matching is case-insensitive.

Example:

```
<analyzer type="query">
  <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SuggestStopFilterFactory" ignoreCase="true"
          words="stopwords.txt" format="wordset"/>
</analyzer>
```

In: "The The"

Tokenizer to Filter: "the"(1), "the"(2)

Out: "the"(2)

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. In the (default) `solr` format - see the `format` argument below for alternatives - 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.

`ignoreCase`: (optional; default: `false`) If `true`, synonyms will be matched case-insensitively.

`expand`: (optional; default: `true`) If `true`, a synonym will be expanded to all equivalent synonyms. If `false`, all equivalent synonyms will be reduced to the first in the list.

`format`: (optional; default: `solr`) Controls how the synonyms will be parsed. The short names `solr` (for `SolrS`

ynonymParser) and wordnet (for WordnetSynonymParser) are supported, or you may alternatively supply the name of your own SynonymMap.Builder subclass.

tokenizerFactory: (optional; default: WhitespaceTokenizerFactory) The name of the tokenizer factory to use when parsing the synonyms file. Arguments with the name prefix "tokenizerFactory." will be supplied as init params to the specified tokenizer factory. Any arguments not consumed by the synonym filter factory, including those without the "tokenizerFactory." prefix, will also be supplied as init params to the tokenizer factory. If tokenizerFactory is specified, then analyzer may not be, and vice versa.

analyzer: (optional; default: WhitespaceTokenizerFactory) The name of the analyzer class to use when parsing the synonyms file. If analyzer is specified, then tokenizerFactory may not be, and vice versa.

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:

`updateOffsets`: if `luceneMatchVersion` is `4.3` or earlier and `updateOffsets="true"`, trimmed tokens' start and end offsets will be updated to those of the first and last characters (plus one) remaining in the token. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

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.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

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" **->** "BugBlaster", "XL". Example 1 below illustrates the default (non-zero) splitting behavior.

`splitOnNumerics`: (integer, default 1) If 0, don't split words on transitions from alpha to numeric:"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" **->** "194732"

`catenateAll`: (0/1, default 0) If non-zero, runs of word and number parts will be joined: "Zap-Master-9000" **->** "ZapMaster9000"

`preserveOriginal`: (integer, default 0) If non-zero, the original token is preserved: "Zap-Master-9000" **->** "Zap-Master-9000", "Zap", "Master", "9000"

`protected`: (optional) The pathname of a file that contains a list of protected words that should be passed 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"

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
- solr.HTMLStripCharFilterFactory
- solr.ICUNormalizer2CharFilterFactory
- solr.PatternReplaceCharFilterFactory
- Related Topics

solr.MappingCharFilterFactory

This filter creates `org.apache.lucene.analysis.MappingCharFilter`, which can be used for changing one string 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>
```

Mapping file syntax:

- Comment lines beginning with a hash mark (#), as well as blank lines, are ignored.
- Each non-comment, non-blank line consists of a mapping of the form: `"source" => "target"`
  - Double-quoted source string, optional whitespace, an arrow (=>), optional whitespace, double-quoted target string.
- Trailing comments on mapping lines are not allowed.
- The source string must contain at least one character, but the target string may be empty.
- The following character escape sequences are recognized within source and target strings:

Escape sequence	Resulting character (ECMA-48 aliases)	Unicode character	Example mapping line
`\\`	`\`	U+005C	`"\\" => "/"`
`\"`	`"`	U+0022	`"\"and\"" => "'and'"`
`\b`	backspace (BS)	U+0008	`"\b" => " "`
`\t`	tab (HT)	U+0009	`"\t" => ","`
`\n`	newline (LF)	U+000A	`"\n" => " "`
`\f`	form feed (FF)	U+000C	`"\f" => "\n"`
`\r`	carriage return (CR)	U+000D	`"\r" => "/carriage-return/"`
`\uXXXX`	Unicode char referenced by the 4 hex digits	U+XXXX	`"\uFEFF" => ""`

  - A backslash followed by any other character is interpreted as if the character were present without the backslash.

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 `A` 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.
- ` ` 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
` 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

Example:

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

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
- KeywordRepeatFilterFactory
- StemmerOverrideFilterFactory
- Dictionary Compound Word Token Filter
- Unicode Collation
- ASCII & Decimal Folding Filters
- Language-Specific Factories
- Related Topics

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 `sample_techproducts_configs` config set 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>
```

KeywordRepeatFilterFactory

Emits each token twice, one with the `KEYWORD` attribute and once without. If placed before a stemmer, the result will be that you will get the unstemmed token preserved on the same position as the stemmed one. Queries matching the original exact term will get a better score while still maintaining the recall benefit of stemming. Another advantage of keeping the original token is that wildcard truncation will work as expected.

To configure, add the `KeywordRepeatFilterFactory` early in the analysis chain. It is recommended to also include `RemoveDuplicatesTokenFilterFactory` to avoid duplicates when tokens are not stemmed.

A sample fieldType configuration could look like this:

```
<fieldtype name="english_stem_preserve_original" class="solr.TextField">
  <analyzer>
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.KeywordRepeatFilterFactory" />
    <filter class="solr.PorterStemFilterFactory" />
    <filter class="solr.RemoveDuplicatesTokenFilterFactory" />
  </analyzer>
</fieldtype>
```



When adding the same token twice, it will also score twice (double), so you may have to re-tune your ranking rules.

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.

`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 8):

`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 8 Collator javadocs for more information.

`decomposition`: Valid values are `no`, `canonical`, or `full`. See Oracle Java 8 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 8 Collator javadocs for more information.

`decomposition`: Valid values are `no`, `canonical`, or `full`. See Oracle Java 8 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 & Decimal Folding Filters

Ascii Folding

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"

Decimal Digit Folding

This filter converts any character in the Unicode "Decimal Number" general category (`"Nd"`) into their equivalent Basic Latin digits (0-9).

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.DecimalDigitFilterFactory`

Arguments: None

Example:

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

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
- CJK
- Czech
- Danish
- Dutch
- Finnish
- French
- Galician
- German
- Greek
- Hebrew, Lao, Myanmar, Khmer
- Hindi
- Indonesian
- Italian
- Irish
- Japanese
- Latvian
- Norwegian
- Persian
- Polish
- Portuguese
- Romanian
- Russian
- Scandinavian

- Serbian
- Spanish
- Swedish
- Thai
- Turkish

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"`.

Also relevant are the Scandinavian normalization filters.

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

Solr can stem Dutch using the Snowball Porter Stemmer with an argument of `language="Dutch"`.

Factory class: `solr.SnowballPorterFilterFactory`

Arguments:

`language`: (required) stemmer language, "Dutch" in this case

Example:

```
<analyzer type="index">
  <tokenizer class="solr.StandardTokenizerFactory"/>
  <filter class="solr.LowerCaseFilterFactory"/>
  <filter class="solr.SnowballPorterFilterFactory" language="Dutch"/>
</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: "haus häuser"

Tokenizer to Filter: "haus", "häuser"

Out: "haus", "haus"

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.IrishLowerCaseFilterFactory`, 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"

Japanese

Solr includes support for analyzing Japanese, via the Lucene Kuromoji morphological analyzer, which includes several analysis components - more details on each below:

- `JapaneseIterationMarkCharFilter` normalizes Japanese horizontal iteration marks (odoriji) to their expanded form.
- `JapaneseTokenizer` tokenizes Japanese using morphological analysis, and annotates each term with part-of-speech, base form (a.k.a. lemma), reading and pronunciation.
- `JapaneseBaseFormFilter` replaces original terms with their base forms (a.k.a. lemmas).
- `JapanesePartOfSpeechStopFilter` removes terms that have one of the configured parts-of-speech.
- `JapaneseKatakanaStemFilter` normalizes common katakana spelling variations ending in a long sound character (U+30FC) by removing the long sound character.

Also useful for Japanese analysis, from lucene-analyzers-common:

- `CJKWidthFilter` folds fullwidth ASCII variants into the equivalent Basic Latin forms, and folds halfwidth Katakana variants into their equivalent fullwidth forms.

Japanese Iteration Mark CharFilter

Normalizes horizontal Japanese iteration marks (odoriji) to their expanded form. Vertical iteration marks are not supported.

Factory class: `JapaneseIterationMarkCharFilterFactory`

Arguments:

`normalizeKanji`: set to `false` to not normalize kanji iteration marks (default is `true`)

`normalizeKana`: set to `false` to not normalize kana iteration marks (default is `true`)

Japanese Tokenizer

Tokenizer for Japanese that uses morphological analysis, and annotates each term with part-of-speech, base form (a.k.a. lemma), reading and pronunciation.

`JapaneseTokenizer` has a `search` mode (the default) that does segmentation useful for search: a heuristic is used to segment compound terms into their constituent parts while also keeping the original compound terms as synonyms.

Factory class: `solr.JapaneseTokenizerFactory`

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 plus unigramming of unknown words (experimental)

  For some applications it might be good to use `search` mode for indexing and `normal` mode for queries to increase precision and prevent parts of compounds from being matched and highlighted.

`userDictionary`: filename for a user dictionary, which allows overriding the statistical model with your own entries for segmentation, part-of-speech tags and readings without a need to specify weights. See `lang/userdict_ja.txt` for a sample user dictionary file.

`userDictionaryEncoding`: user dictionary encoding (default is UTF-8)

`discardPunctuation`: set to `false` to keep punctuation, `true` to discard (the default)

Japanese Base Form Filter

Replaces original terms' text with the corresponding base form (lemma). (`JapaneseTokenizer` annotates each term with its base form.)

Factory class: `JapaneseBaseFormFilterFactory`

(no arguments)

Japanese Part Of Speech Stop Filter

Removes terms with one of the configured parts-of-speech. `JapaneseTokenizer` annotates terms with parts-of-speech.

Factory class : `JapanesePartOfSpeechStopFilterFactory`

Arguments:

`tags`: filename for a list of parts-of-speech for which to remove terms; see `conf/lang/stoptags_ja.txt` in the `sample_techproducts_config` config set for an example.

`enablePositionIncrements`: if `luceneMatchVersion` is `4.3` or earlier and `enablePositionIncrements="false"`, no position holes will be left by this filter when it removes tokens. **This argument is invalid if `luceneMatchVersion` is `5.0` or later.**

Japanese Katakana Stem Filter

Normalizes common katakana spelling variations ending in a long sound character (U+30FC) by removing the long sound character.

`CJKWidthFilterFactory` should be specified prior to this filter to normalize half-width katakana to full-width.

Factory class: `JapaneseKatakanaStemFilterFactory`

Arguments:

`minimumLength`: terms below this length will not be stemmed. Default is 4, value must be 2 or more.

CJK Width Filter

Folds fullwidth ASCII variants into the equivalent Basic Latin forms, and folds halfwidth Katakana variants into their equivalent fullwidth forms.

Factory class: `CJKWidthFilterFactory`

(no arguments)

Example:

```
<fieldType name="text_ja" positionIncrementGap="100"
autoGeneratePhraseQueries="false">
  <analyzer>
    <!-- Uncomment if you need to handle iteration marks: -->
    <!-- <charFilter class="solr.JapaneseIterationMarkCharFilterFactory" /> -->
    <tokenizer class="solr.JapaneseTokenizerFactory" mode="search"
userDictionary="lang/userdict_ja.txt"/>
    <filter class="solr.JapaneseBaseFormFilterFactory"/>
    <filter class="solr.JapanesePartOfSpeechStopFilterFactory"
tags="lang/stoptags_ja.txt"/>
    <filter class="solr.CJKWidthFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true"
words="lang/stopwords_ja.txt"/>
    <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".

Also relevant are the Scandinavian normalization filters.

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"/>
    <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"/>
    <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="morfologik/stemming/polish/polish.dict"/>
</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 param value is a constant specifying which dictionary to choose. The dictionary resource must be named `morfologik/stemming/`*`language`*`/`*`language`*`.dict` and have an associated `.info` metadata file. See the Morfologik project for details. If the dictionary attribute is not provided, the Polish dictionary is loaded and used by default.

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>
```

Scandinavian

Scandinavian is a language group spanning three languages Norwegian, Swedish and Danish which are very similar.

Swedish å,ä,ö are in fact the same letters as Norwegian and Danish å,æ,ø and thus interchangeable when used between these languages. They are however folded differently when people type them on a keyboard lacking these characters.

In that situation almost all Swedish people use a, a, o instead of å, ä, ö. Norwegians and Danes on the other hand usually type aa, ae and oe instead of å, æ and ø. Some do however use a, a, o, oo, ao and sometimes permutations of everything above.

There are two filters for helping with normalization between Scandinavian languages: one is `solr.ScandinavianNormalizationFilterFactory` trying to preserve the special characters (æäöå) and another `solr.ScandinavianFoldingFilterFactory` which folds these to the more broad ø/ö->o etc.

See also each language section for other relevant filters.

Scandinavian Normalization Filter

This filter normalize use of the interchangeable Scandinavian characters æÆäÄöÖøØ and folded variants (aa, ao, ae, oe and oo) by transforming them to åÅæÆøØ.

It's a semantically less destructive solution than `ScandinavianFoldingFilter`, most useful when a person with a Norwegian or Danish keyboard queries a Swedish index and vice versa. This filter does **not** perform the common Swedish folds of å and ä to a nor ö to o.

Factory class: `solr.ScandinavianNormalizationFilterFactory`

Arguments: None

Example:

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

In: "blåbærsyltetøj blåbärsyltetöj blaabaarsyltetoej blabarsyltetoj"

Tokenizer to Filter: "blåbærsyltetøj", "blåbärsyltetöj", "blaabaersyltetoej", "blabarsyltetoj"

Out: "blåbærsyltetøj", "blåbærsyltetøj", "blåbærsyltetøj", "blabarsyltetoj"

Scandinavian Folding Filter

This filter folds Scandinavian characters åÅäæÄÆ->a and öÖøØ->o. It also discriminate against use of double vowels aa, ae, ao, oe and oo, leaving just the first one.

It's is a semantically more destructive solution than `ScandinavianNormalizationFilter`, but can in addition help with matching raksmorgas as räksmörgås.

Factory class: `solr.ScandinavianFoldingFilterFactory`

Arguments: None

Example:

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

In: "blåbærsyltetøj blåbärsyltetöj blaabaarsyltetoej blabarsyltetoj"

Tokenizer to Filter: "blåbærsyltetøj", "blåbärsyltetöj", "blaabaersyltetoej", "blabarsyltetoj"

Out: "blabarsyltetoj", "blabarsyltetoj", "blabarsyltetoj", "blabarsyltetoj"

Serbian

Serbian Normalization Filter

Solr includes a filter that normalizes Serbian Cyrillic and Latin characters. Note that this filter only works with lowercased input.

See the Solr wiki for tips & advice on using this filter: https://wiki.apache.org/solr/SerbianLanguageSupport

Factory class: `solr.SerbianNormalizationFilterFactory`

Arguments: `haircut` : Select the extend of normalization. Valid values are:

- bald: (Default behavior) Cyrillic characters are first converted to Latin; then, Latin characters have their diacritics removed, with the exception of "LATIN SMALL LETTER D WITH STROKE" (U+0111) which is converted to "`dj`"
- `regular`: Only Cyrillic to Latin normalization will be applied, preserving the Latin diatrics

Example:

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

Also relevant are the Scandinavian normalization filters.

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

Phonetic matching algorithms may be used to encode tokens so that two different spellings that are pronounced similarly will match.

For overviews of and comparisons between algorithms, see http://en.wikipedia.org/wiki/Phonetic_algorithm and http://ntz-develop.blogspot.com/2011/03/phonetic-algorithms.html

Algorithms discussed in this section:

- Beider-Morse Phonetic Matching (BMPM)
- Daitch-Mokotoff Soundex
- Double Metaphone
- Metaphone
- Soundex
- Refined Soundex
- Caverphone
- Kölner Phonetik a.k.a. Cologne Phonetic
- NYSIIS

Beider-Morse Phonetic Matching (BMPM)

To use this encoding in your analyzer, see Beider Morse Filter in the Filter Descriptions section.

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
- 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.

Daitch-Mokotoff Soundex

To use this encoding in your analyzer, see Daitch-Mokotoff Soundex Filter in the Filter Descriptions section.

The Daitch-Mokotoff Soundex algorithm is a refinement of the Russel and American Soundex algorithms, yielding greater accuracy in matching especially Slavic and Yiddish surnames with similar pronunciation but differences in spelling.

The main differences compared to the other soundex variants are:

- coded names are 6 digits long
- initial character of the name is coded
- rules to encoded multi-character n-grams

- multiple possible encodings for the same name (branching)

Note: the implementation used by Solr (commons-codec's `DaitchMokotoffSoundex`) has additional branching rules compared to the original description of the algorithm.

For more information, see http://en.wikipedia.org/wiki/Daitch%E2%80%93Mokotoff_Soundex and http://www.avotaynu.com/soundex.htm

Double Metaphone

To use this encoding in your analyzer, see Double Metaphone Filter in the Filter Descriptions section. Alternatively, you may specify `encoding="DoubleMetaphone"` with the Phonetic Filter, but note that the Phonetic Filter version will **not** provide the second ("alternate") encoding that is generated by the Double Metaphone Filter for some tokens.

Encodes tokens using the double metaphone algorithm by Lawrence Philips. See the original article at http://www.drdobbs.com/the-double-metaphone-search-algorithm/184401251?pgno=2

Metaphone

To use this encoding in your analyzer, specify `encoding="Metaphone"` with the Phonetic Filter.

Encodes tokens using the Metaphone algorithm by Lawrence Philips, described in "Hanging on the Metaphone" in Computer Language, Dec. 1990.

See http://en.wikipedia.org/wiki/Metaphone

Soundex

To use this encoding in your analyzer, specify `encoding="Soundex"` with the Phonetic Filter.

Encodes tokens using the Soundex algorithm, which is used to relate similar names, but can also be used as a general purpose scheme to find words with similar phonemes.

See http://en.wikipedia.org/wiki/Soundex

Refined Soundex

To use this encoding in your analyzer, specify `encoding="RefinedSoundex"` with the Phonetic Filter.

Encodes tokens using an improved version of the Soundex algorithm.

See http://en.wikipedia.org/wiki/Soundex

Caverphone

To use this encoding in your analyzer, specify `encoding="Caverphone"` with the Phonetic Filter.

Caverphone is an algorithm created by the Caversham Project at the University of Otago. The algorithm is optimised for accents present in the southern part of the city of Dunedin, New Zealand.

See http://en.wikipedia.org/wiki/Caverphone and the Caverphone 2.0 specification at http://caversham.otago.ac.nz/files/working/ctp150804.pdf

Kölner Phonetik a.k.a. Cologne Phonetic

To use this encoding in your analyzer, specify `encoding="ColognePhonetic"` with the Phonetic Filter.

The Kölner Phonetik, an algorithm published by Hans Joachim Postel in 1969, is optimized for the German language.

See http://de.wikipedia.org/wiki/K%C3%B6lner_Phonetik

NYSIIS

To use this encoding in your analyzer, specify `encoding="Nysiis"` with the Phonetic Filter.

NYSIIS is an encoding used to relate similar names, but can also be used as a general purpose scheme to find words with similar phonemes.

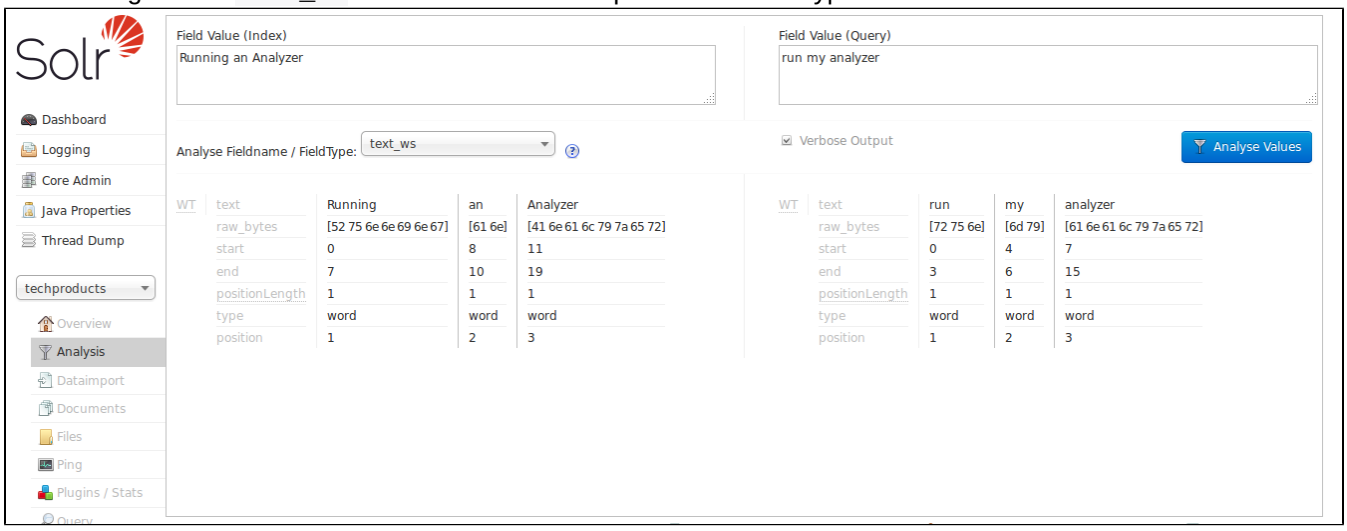
See http://en.wikipedia.org/wiki/NYSIIS and http://www.dropby.com/NYSIIS.html

Running Your Analyzer

Once you've defined a field type in your Schema, 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, let's look at some of the "Text" field types available in the "`bin/solr -e techproducts`" example configuration, and use the Analysis Screen (http://localhost:8983/solr/#/techproducts/analysis) to compare how the tokens produced at index time for the sentence "`Running an Analyzer`" match up with a slightly different query text of "`run my analyzer`"

We can begin with "`text_ws`" - one of the most simplified Text field types available:



By looking at the start and end positions for each term, we can see that the only thing this field type does is tokenize text on whitespace. Notice in this image that the term "Running" has a start position of 0 and an end position of 7, while "an" has a start position of 8 and an end position of 10, and "Analyzer" starts at 11 and ends at 19. If the whitespace between the terms was also included, the count would be 21; since it is 19, we know that whitespace has been removed from this query.

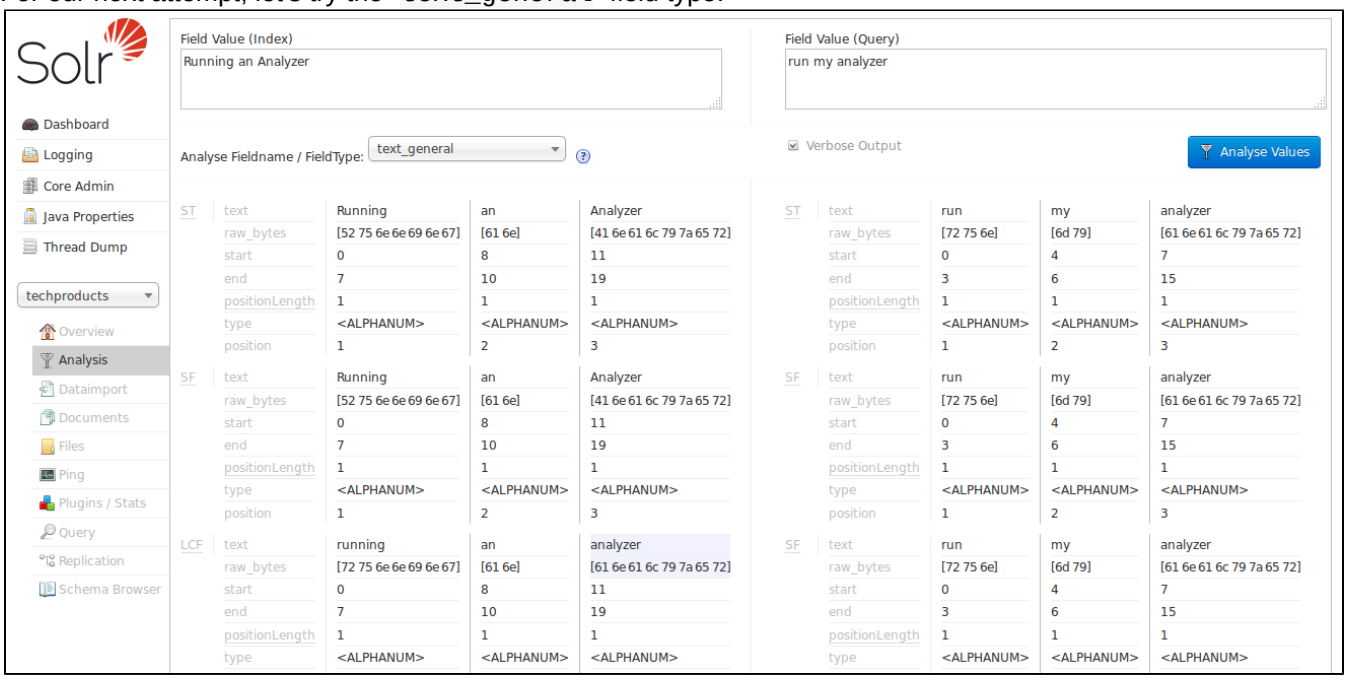
Note also that the indexed terms and the query terms are still very different. "Running" doesn't match "run", "Analyzer" doesn't match "analyzer" (to a computer), and obviously "an" and "my" are totally different words. If

our objective is to allow queries like "`run my analyzer`" to match indexed text like "`Running an Analyzer`" then we will evidently need to pick a different field type with index & query time text analysis that does more processing of the inputs.

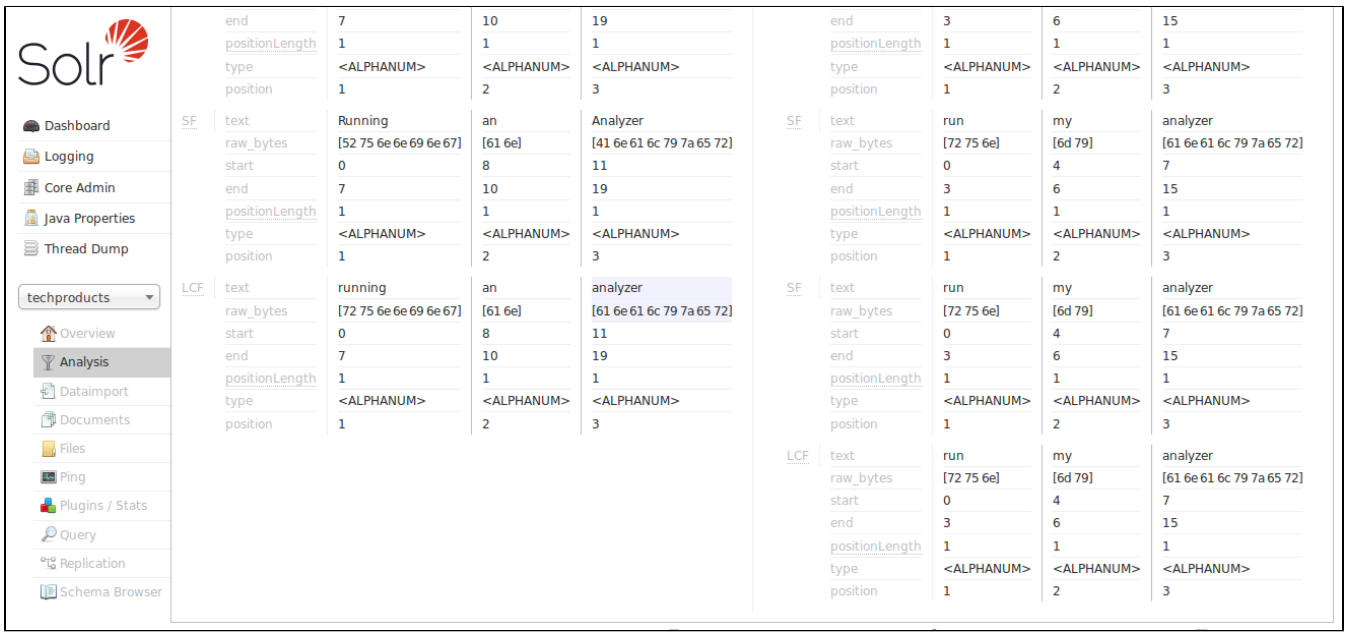
In particular we will want:

- Case insensitivity, so "Analyzer" and "analyzer" match.
- Stemming, so words like "Run" and "Running" are considered equivalent terms.
- Stop Word Pruning, so small words like "an" and "my" don't affect the query.

For our next attempt, let's try the "`text_general`" field type:

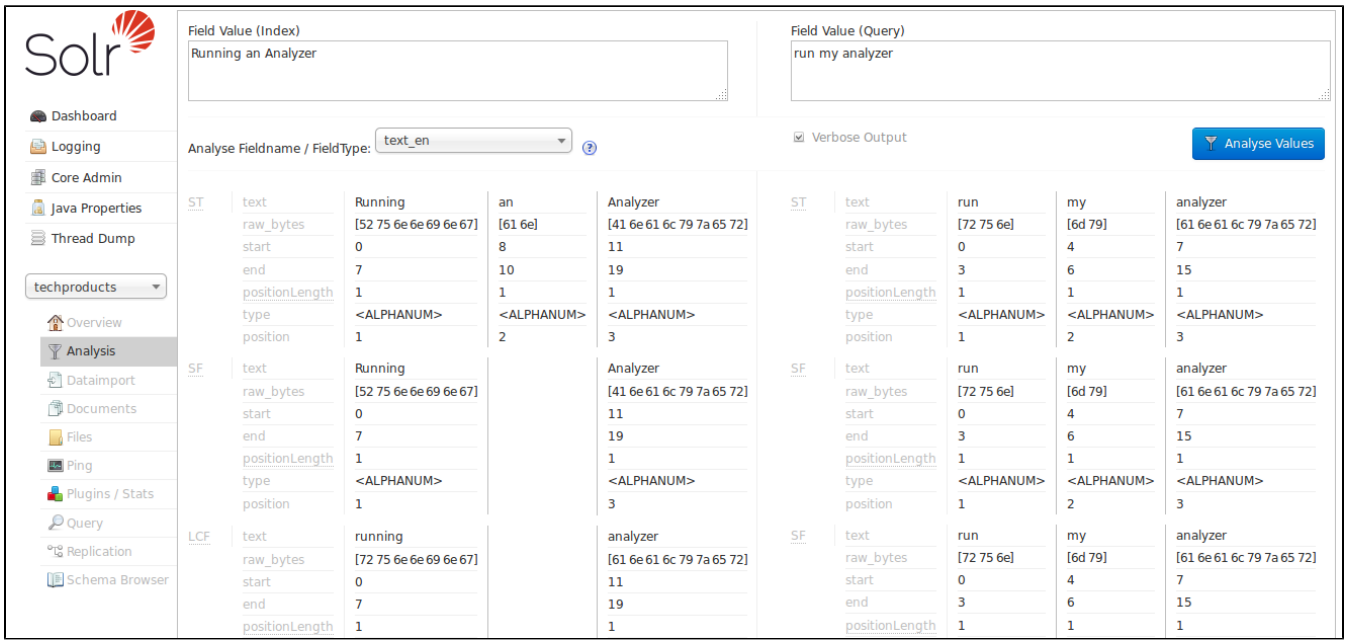


With the verbose output enabled, we can see how each stage of our new analyzers modify the tokens they receive before passing them on to the next stage. As we scroll down to the final output, we can see that we do start to get a match on "analyzer" from each input string, thanks to the "LCF" stage -- which if you hover over with your mouse, you'll see is the "`LowerCaseFilter`":

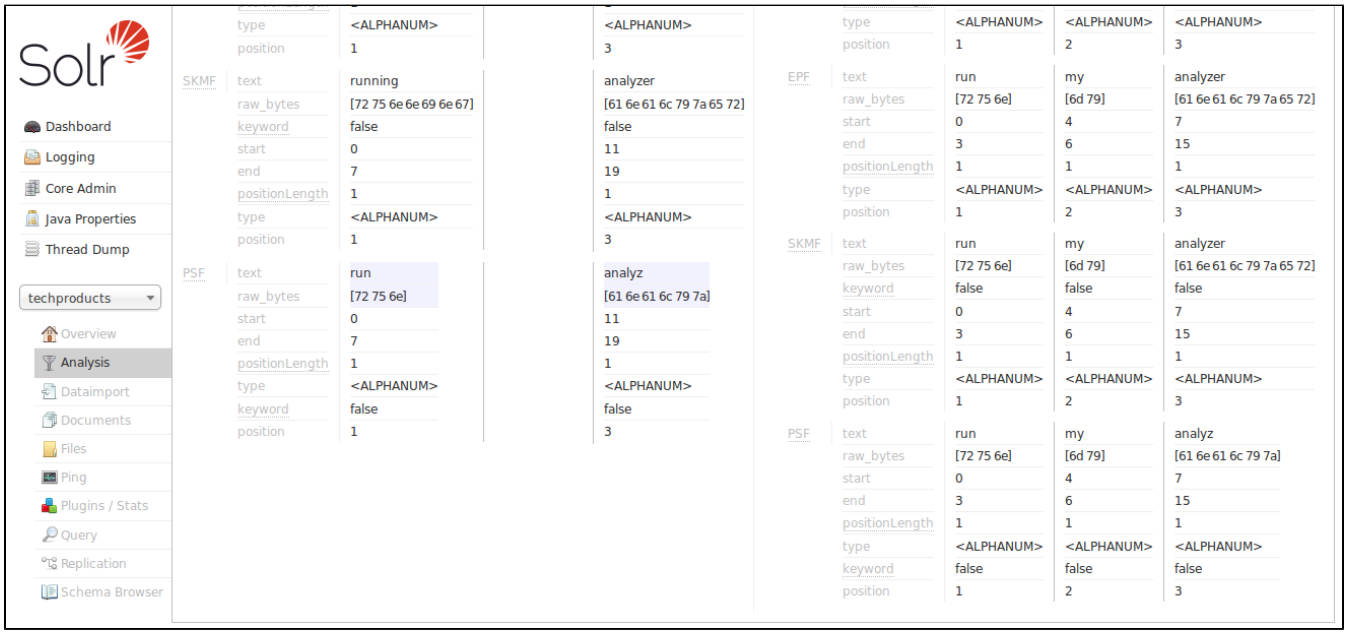


The "`text_general`" field type is designed to be generally useful for any language, and it has definitely gotten us closer to our objective than "`text_ws`" from our first example by solving the problem of case sensitivity. It's still not quite what we are looking for because we don't see stemming or stopword rules being applied.

So now let us try the "`text_en`" field type:



Now we can see the "SF" (`StopFilter`) stage of the analyzers solving the problem of removing Stop Words ("an"), and as we scroll down, we also see the "PSF" (`PorterStemFilter`) stage apply stemming rules suitable for our English language input, such that the terms produced by our "index analyzer" and the terms produced by our "query analyzer" match the way we expect.



At this point, we can continue to experiment with additional inputs, verifying that our analyzers produce matching tokens when we expect them to match, and disparate tokens when we do not expect them to match, as we

iterate and tweak our field type configuration.

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.
- **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 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

When starting Solr with the "-e" option, the `example/` directory will be used as base directory for the example Solr instances that are created. This directory also includes an `example/exampledocs/` subdirectory containing sample documents in a variety of formats that you can use to experiment with indexing into the various examples.

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.

Post Tool

Solr includes a simple command line tool for POSTing various types of content to a Solr server. The tool is `bin/post`. The bin/post tool is a Unix shell script; for Windows (non-Cygwin) usage, see the Windows section below.

To run it, open a window and enter:

```
bin/post -c gettingstarted example/films/films.json
```

This will contact the server at `localhost:8983.` Specifying the `collection/core name` is **mandatory**. The '-help' (or simply '-h') option will output information on its usage (i.e., `bin/post -help`).

Using the bin/post Tool

Specifying either the `collection/core name` or the full update `url` is **mandatory** when using `bin/post`.

The basic usage of `bin/post` is:

```
$ bin/post -h
Usage: post -c <collection> [OPTIONS] <files|directories|urls|-d ["...",...]>
    or post -help

   collection name defaults to DEFAULT_SOLR_COLLECTION if not specified

OPTIONS
=======
  Solr options:
    -url <base Solr update URL> (overrides collection, host, and port)
    -host <host> (default: localhost)
    -p or -port <port> (default: 8983)
    -commit yes|no (default: yes)

  Web crawl options:
    -recursive <depth> (default: 1)
    -delay <seconds> (default: 10)


  Directory crawl options:
    -delay <seconds> (default: 0)

  stdin/args options:
    -type <content/type> (default: application/xml)


  Other options:
    -filetypes <type>[,<type>,...] (default:
xml,json,csv,pdf,doc,docx,ppt,pptx,xls,xlsx,odt,odp,ods,ott,otp,ots,rtf,htm,html,txt
,log)
    -params "<key>=<value>[&<key>=<value>...]" (values must be URL-encoded; these
pass through to Solr update request)
    -out yes|no (default: no; yes outputs Solr response to console)
...
```

Examples

There are several ways to use `bin/post`. This section presents several examples.

Indexing XML

Add all documents with file extension `.xml` to collection or core named `gettingstarted`.

```
bin/post -c gettingstarted *.xml
```

Add all documents with file extension `.xml` to the `gettingstarted` collection/core on Solr running on port `8984`.

```
bin/post -c gettingstarted -p 8984 *.xml
```

Send XML arguments to delete a document from `gettingstarted`.

```
bin/post -c gettingstarted -d '<delete><id>42</id></delete>'
```

Indexing CSV

Index all CSV files into `gettingstarted`:

```
bin/post -c gettingstarted *.csv
```

Index a tab-separated file into `gettingstarted`:

```
bin/post -c signals -params "separator=%09" -type text/csv data.tsv
```

The content type (`-type`) parameter is required to treat the file as the proper type, otherwise it will be ignored and a WARNING logged as it does not know what type of content a .tsv file is. The CSV handler supports the `separator` parameter, and is passed through using the `-params` setting.

Indexing JSON

Index all JSON files into `gettingstarted`.

```
bin/post -c gettingstarted *.json
```

Indexing rich documents (PDF, Word, HTML, etc)

Index a PDF file into `gettingstarted`.

```
bin/post -c gettingstarted a.pdf
```

Automatically detect content types in a folder, and recursively scan it for documents for indexing into `gettingstarted`.

```
bin/post -c gettingstarted afolder/
```

Automatically detect content types in a folder, but limit it to PPT and HTML files and index into `gettingstarted`.

```
bin/post -c gettingstarted -filetypes ppt,html afolder/
```

Windows support

`bin/post` exists currently only as a Unix shell script, however it delegates its work to a cross-platform capable Java program. The `SimplePostTool` can be run directly in supported environments, including Windows.

SimplePostTool

The `bin/post` script currently delegates to a standalone Java program called `SimplePostTool`. This tool, bundled into a executable JAR, can be run directly using `java -jar example/exampledocs/post.jar`. See the help output and take it from there to post files, recurse a website or file system folder, or send direct commands to a Solr server.

```
$ java -jar example/exampledocs/post.jar -h
SimplePostTool version 5.0.0
Usage: java [SystemProperties] -jar post.jar [-h|-] [<file|folder|url|arg>
[<file|folder|url|arg>...]]
.
.
.
```

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 and use request handlers is with path based names that map to paths in the request url. However, request handlers can also be specified with the `qt` (query type) parameter if the `requestDispatcher` is appropriately configured. 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.

A single unified update request handler supports XML, CSV, JSON, and javabin update requests, delegating to the appropriate `ContentStreamLoader` based on the `Content-Type` of the ContentStream.

Topics covered in this section:

- UpdateRequestHandler Configuration
- XML Formatted Index Updates
  - Adding Documents
  - XML Update Commands
  - Using curl to Perform Updates
  - Using XSLT to Transform XML Index Updates
- JSON Formatted Index Updates
  - Solr-Style JSON
  - JSON Update Convenience Paths
  - Transforming and Indexing Custom JSON
- CSV Formatted Index Updates
  - CSV Update Parameters
  - Indexing Tab-Delimited files
  - CSV Update Convenience Paths
- Nested Child Documents

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.

XML Update Commands

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>
```



When using the Join query parser in a Delete By Query, you should use the `score` parameter with a value of "`none`" to avoid a `ClassCastException`. See the section on the Join Query Parser for more details on the `score` parameter.

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

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/solr/my_collection/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/solr/my_collection/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/solr/my_collection/update?stream.body=%3Ccommit/%3E
```

Responses from Solr take the form shown here:

```
<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.

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 `conf/xslt` directory of your config set 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 version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:output media-type="text/xml" method="xml" indent="yes"/>
  <xsl:template match='/'>
    <add>
      <xsl:apply-templates select="response/result/doc"/>
    </add>
  </xsl:template>
  <!-- Ignore score (makes no sense to index) -->
  <xsl:template match="doc/*[@name='score']" priority="100"></xsl:template>
  <xsl:template match="doc">
    <xsl:variable name="pos" select="position()"/>
    <doc>
      <xsl:apply-templates>
        <xsl:with-param name="pos"><xsl:value-of select="$pos"/></xsl:with-param>
      </xsl:apply-templates>
    </doc>
  </xsl:template>
  <!-- Flatten arrays to duplicate field lines -->
  <xsl:template match="doc/arr" priority="100">
    <xsl:variable name="fn" select="@name"/>
    <xsl:for-each select="*">
      <xsl:element name="field">
        <xsl:attribute name="name"><xsl:value-of select="$fn"/></xsl:attribute>
        <xsl:value-of select="."/>
      </xsl:element>
    </xsl:for-each>
  </xsl:template>
  <xsl:template match="doc/*">
    <xsl:variable name="fn" select="@name"/>
      <xsl:element name="field">
        <xsl:attribute name="name"><xsl:value-of select="$fn"/></xsl:attribute>
      <xsl:value-of select="."/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*"/>
</xsl:stylesheet>
```

This stylesheet transforms Solr's XML search result format into Solr's Update XML syntax. One example usage would be to copy a Solr 1.3 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/my_collection/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/my_collection/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

Solr can accept JSON that conforms to a defined structure, or can accept arbitrary JSON-formatted documents. If sending arbitrarily formatted JSON, there are some additional parameters that need to be sent with the update request, described below in the section Transforming and Indexing Custom JSON.

Solr-Style JSON

JSON formatted update requests may be sent to Solr's `/update` handler using `Content-Type: application/json` or `Content-Type: text/json`.

JSON formatted updates can take 3 basic forms, described in depth below:

- 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.
- A list of documents to add, expressed as a top level JSON Array containing a JSON Object per document.
- A sequence of update commands, expressed as a top level JSON Object (aka: Map).

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 `/update/json/docs` path:

```
curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/my_collection/update/json/docs' --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/my_collection/update' --data-binary '
[
  {
    "id": "1",
    "title": "Doc 1"
  },
  {
    "id": "2",
    "title": "Doc 2"
  }
]'
```

A sample JSON file is provided at `example/exampledocs/books.json` and contains an array of objects that you can add to the Solr `techproducts` example:

```
curl 'http://localhost:8983/solr/techproducts/update?commit=true' --data-binary
@example/exampledocs/books.json -H 'Content-type:application/json'
```

Sending JSON Update Commands

In general, the JSON update syntax supports 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/my_collection/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, a single document:

```
{ "delete":"myid" }
```

Or a list of document IDs:

```
{ "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` `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.

Transforming and Indexing Custom JSON

If you have JSON documents that you would like to index without transforming them into Solr's structure, you can add them to Solr by including some parameters with the update request. These parameters provide information on how to split a single JSON file into multiple Solr documents and how to map fields to Solr's schema. One or more valid JSON documents can be sent to the `/update/json/docs` path with the configuration params.

Mapping Parameters

These parameters allow you to define how a JSON file should be read for multiple Solr documents.

- **split**: Defines the path at which to split the input JSON into multiple Solr documents and is required if you have multiple documents in a single JSON file. If the entire JSON makes a single solr document, the path must be "`/`". It is possible to pass multiple split paths by separating them with a pipe (`|`) example : `split=/|/foo|/foo/bar` . If one path is a child of another, they automatically become a child document
  f: This is a multivalued mapping parameter. The format of the parameter is `target-field-name:json-path`. The `json-path` is required. The `target-field-name` is the Solr document field name, and is optional. If not specified, it is automatically derived from the input JSON.The default target field name is the fully qualified name of the field. Wildcards can be used here, see the section Wildcards below for more information.
- **mapUniqueKeyOnly** (boolean): This parameter is particularly convenient when the fields in the input JSON are not available in the schema and schemaless mode is not enabled. This will index all the fields into the default search field (using the `df` parameter, below) and only the `uniqueKey` field is mapped to the corresponding field in the schema. If the input JSON does not have a value for the `uniqueKey` field then a UUID is generated for the same.
- **df**: If the `mapUniqueKeyOnly` flag is used, the update handler needs a field where the data should be

indexed to. This is the same field that other handlers use as a default search field.

- **srcField**: This is the name of the field to which the JSON source will be stored into. This can only be used if `split=/` (i.e., you want your JSON input file to be indexed as a single Solr document). Note that atomic updates will cause the field to be out-of-sync with the document.
- **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.

For example, if we have a JSON file that includes two documents, we could define an update request like this:

```
curl 'http://localhost:8983/solr/my_collection/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}
  ]
}'
```

With this request, we have defined that "exams" contains multiple documents. In addition, we have mapped several fields from the input document to Solr fields.

When the update request is complete, the following two documents will be added to the index:

```
{
  "first":"John",
  "last":"Doe",
  "marks":90,
  "test":"term1",
  "subject":"Maths",
  "grade":8
}
{
  "first":"John",
  "last":"Doe",
  "marks":86,
  "test":"term1",
  "subject":"Biology",
  "grade":8
}
```

In the prior example, all of the fields we wanted to use in Solr had the same names as they did in the input

JSON. When that is the case, we can simplify the request as follows:

```
curl 'http://localhost:8983/solr/my_collection/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}
  ]
}'
```

In this example, we simply named the field paths (such as `/exams/test`). Solr will automatically attempt to add the content of the field from the JSON input to the index in a field with the same name.

Note that if you are not working in Schemaless Mode, where fields that don't exist will be created on the fly with Solr's best guess for the field type, documents may get rejected if the fields do not exist in the schema before indexing.

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=$FQN:/**`: maps all fields to the fully qualified name (`$FQN`) of the JSON field. The fully qualified name is obtained by concatenating all the keys in the hierarchy with a period (`.`) as a delimiter. This is the default behavior if no `f` path mappings are specified.
- `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 further simplify our previous example as follows:

```
curl 'http://localhost:8983/solr/my_collection/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}
  ]
}'
```

Because we want the fields to be indexed with the field names as they are found in the JSON input, the double wildcard in `f=/**` will map all fields and their descendants to the same fields in Solr.

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.

```
curl 'http://localhost:8983/solr/my_collection/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}
  ]
}'
```

In the above example, we've said all of the fields should be added to a field in Solr named 'txt'. This will add multiple fields to a single field, so whatever field you choose should be multi-valued.

The default behavior is to use the fully qualified name (FQN) of the node. So, if we don't define any field mappings, like this:

```
curl 'http://localhost:8983/solr/my_collection/update/json/docs?split=/exams'\
    -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}
  ]
}'
```

The indexed documents would be added to the index with fields that look like this:

```
{
  "first":"John",
  "last":"Doe",
  "grade":8,
  "exams.subject":"Maths",
  "exams.test":"term1",
  "exams.marks":90},
{
  "first":"John",
  "last":"Doe",
  "grade":8,
  "exams.subject":"Biology",
  "exams.test":"term1",
  "exams.marks":86}
```

Indexing nested docs

The following is an example of indexing nested docs,

```
curl 'http://localhost:8983/solr/my_collection/update/json/docs?split=/|/orgs'\
    -H 'Content-type:application/json' -d '{
  "name": "Joe Smith",
  "phone": 876876687,
  "orgs": [
    {
      "name": "Microsoft",
      "city": "Seattle",
      "zip": 98052
    },
    {
      "name": "Apple",
      "city": "Cupertino",
      "zip": 95014
    }
  ]
}'
```

the docs indexed would be,

```
{
  "name":"Joe Smith",
  "phone":876876687,
  "_childDocuments_":[
    {
      "name":"Microsoft",
      "city":"Seattle",
      "zip":98052},
    {
      "name":"Apple",
      "city":"Cupertino",
      "zip":95014}]}
```

Setting JSON Defaults

It is possible to send any json to the `/update/json/docs` endpoint and the default configuration of the component is as follows:

```
<initParams path="/update/json/docs">
  <lst name="defaults">
    <!-- this ensures that the entire json doc will be stored verbatim into one
field -->
    <str name="srcField">_src_</str>
    <!-- This means a the uniqueKeyField will be extracted from the fields and
         all fields go into the 'df' field. In this config df is already configured
to be 'text'
     -->
    <str name="mapUniqueKeyOnly">true</str>
    <!-- The default search field where all the values are indexed to -->
    <str name="df">text</str>
  </lst>
</initParams>
```

So, if no params are passed, the entire json file would get indexed to the `_src_` field and all the values in the input JSON would go to a field named `text`. If there is a value for the uniqueKey it is stored and if no value

could be obtained from the input JSON, a UUID is created and used as the uniqueKey field value.

CSV Formatted Index Updates

CSV formatted update requests may be sent to Solr's `/update` handler using `Content-Type: application/csv` or `Content-Type: text/csv`.

A sample CSV file is provided at `example/exampledocs/books.csv` that you can use to add some documents to the Solr `techproducts` example:

```
curl 'http://localhost:8983/solr/techproducts/update?commit=true' --data-binary
@example/exampledocs/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=%09
trim	If true, remove leading and trailing whitespace from values. Default=false.	g,f	f.isbn.trim=true trim=false
header	Set to true if first line of input contains field names. These will be used if the **fieldnames** par ameter is absent.	g	
fieldnames	Comma separated list of field names to use when adding documents.	g	fieldnames=isbn,price,title
literal.<field_name>	A literal value for a specified field name.	g	literal.color=red
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 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

Indexing Tab-Delimited files

The same feature used to index CSV documents can also be easily used to index tab-delimited files (TSV files) and even handle backslash escaping rather than CSV encapsulation.

For example, one can dump a MySQL table to a tab delimited file with:

```
SELECT * INTO OUTFILE '/tmp/result.txt' FROM mytable;
```

This file could then be imported into Solr by setting the `separator` to tab (%09) and the `escape` to backslash (%5c).

```
curl 'http://localhost:8983/solr/update/csv?commit=true&separator=%09&escape=%5c'
--data-binary @/tmp/result.txt
```

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 indexes nested documents in blocks 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 can search 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">New Lucene and Solr release 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": "New Lucene and Solr release is out",
    "content_type": "parentDocument",
    "_childDocuments_": [
      {
        "id": "4",
        "comments": "Lots of new features"
      }
    ]
  }
]
```

Note
One limitation of indexing nested documents is that the whole block of parent-children documents must be updated together whenever any changes are required. In other words, even if a single child document or the parent document is changed, the whole block of parent-child documents must be indexed together.

Uploading Data with Solr Cell using Apache Tika

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.

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.

Topics covered in this section:

- Key Concepts
- Trying out Tika with the Solr techproducts Example
- Input Parameters
- Order of Operations
- Configuring the Solr ExtractingRequestHandler
- Indexing Encrypted Documents with the ExtractingUpdateRequestHandler
- Examples
- Sending Documents to Solr with a POST
- Sending Documents to Solr with Solr Cell and SolrJ
- Related Topics

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/1.7/formats.html for the file types supported.
- Tika adds all the extracted text to the `content` field.
- You can map Tika's metadata fields to Solr fields. You can also boost these fields.
- You can pass in literals for field values. 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 `techproducts` Example

You can try out the Tika framework using the `techproducts` example included in Solr.

Start the example:

```
bin/solr -e techproducts
```

You can now use curl to send a sample PDF file via HTTP POST:

```
curl
'http://localhost:8983/solr/techproducts/update/extract?literal.id=doc1&commit=true'
-F "myfile=@example/exampledocs/solr-word.pdf"
```

The URL above calls the Extracting Request Handler, uploads the file `solr-word.pdf` 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.

You can also use `bin/post` to send a PDF file into Solr (without the params, the literal.id parameter would be set to the absolute path to the file):

```
bin/post -c techproducts example/exampledocs/solr-word.pdf -params "literal.id=a"
```

Now you should be able to execute a query and find that document. You can make a request like `http://localhost:8983/solr/techproducts/select?q=pdf`.

You may notice that although the content of the sample document has been indexed and stored, there are not a lot of metadata fields associated with this document. This is because unknown fields are ignored according to the default parameters configured for the `/update/extract` handler in `solrconfig.xml`, and this behavior can be easily changed or overridden. For example, to store and see all metadata and content, execute the following:

```
bin/post -c techproducts example/exampledocs/solr-word.pdf -params
"literal.id=doc1&uprefix=attr_"
```

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 and indexed.

This command allows you to query the document using an attribute, as in: `http://localhost:8983/solr/techproducts/select?q=attr_meta:microsoft`.

Input Parameters

The table below describes the parameters accepted by the Extracting 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.
ignoreTikaException	If true, exceptions found during processing will be skipped. Any metadata available, however, will be indexed.
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/1.7/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 Extracting 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 with the supplied `sample_techproducts_configs` or `data_driven_schema_configs` config set, you must configure your own `solrconfig.xml` to know about the Jar's containing the `ExtractingRequestHandler` and it's dependencies:

```
<lib dir="${solr.install.dir:../../..}/contrib/extraction/lib" regex=".*\.jar" />
  <lib dir="${solr.install.dir:../../..}/dist/" regex="solr-cell-\d.*\.jar" />
```

You can then configure 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>
  <!-- Optional. Specify an external file containing parser-specific properties.
       This file is located in the same directory as solrconfig.xml by default.-->
  <str name="parseContext.config">parseContext.xml</str>
</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" />
  ...
```

Parser specific properties

Parsers used by Tika may have specific properties to govern how data is extracted. For instance, when using the Tika library from a Java program, the PDFParserConfig class has a method setSortByPosition(boolean) that can extract vertically oriented text. To access that method via configuration with the ExtractingRequestHandler, one can add the parseContext.config property to the solrconfig.xml file (see above) and then set properties in Tika's PDFParserConfig as below. Consult the Tika Java API documentation for configuration parameters that can be set for any particular parsers that require this level of control.

```
<entries>
  <entry class="org.apache.tika.parser.pdf.PDFParserConfig"
impl="org.apache.tika.parser.pdf.PDFParserConfig">
    <property name="extractInlineImages" value="true"/>
    <property name="sortByPosition" value="true"/>
  </entry>
  <entry>...</entry>
</entries>
```

Multi-Core Configuration

For a multi-core configuration, you can specify `sharedLib='lib'` in the `<solr/>` section of `solr.xml` and place the necessary jar files there.

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 Extracting 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`.

```
bin/post -c techproducts example/exampledocs/sample.html -params
"literal.id=doc2&captureAttr=true&defaultField=_text_&fmap.div=foo_t&capture=div"
```

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.

```
bin/post -c techproducts example/exampledocs/sample.html -params
"literal.id=doc3&captureAttr=true&defaultField=_text_&capture=div&fmap.div=foo_t&boo
st.foo_t=3"
```

Using Literals to Define Your Own Metadata

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

```
bin/post -c techproducts -params
"literal.id=doc4&captureAttr=true&defaultField=text&capture=div&fmap.div=foo_t&boost
.foo_t=3&literal.blah_s=Bah" example/exampledocs/sample.html
```

XPath

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

```
bin/post -c techproducts -params
"literal.id=doc5&captureAttr=true&defaultField=text&capture=div&fmap.div=foo_t&boost
.foo_t=3&xpath=/xhtml:html/xhtml:body/xhtml:div//node()"
example/exampledocs/sample.html
```

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/techproducts/update/extract?&extractOnly=true"
--data-binary @example/exampledocs/sample.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 (`-out yes` instructs the tool to echo Solr's output to the console):

```
bin/post -c techproducts -params "extractOnly=true&wt=ruby&indent=true" -out yes
example/exampledocs/sample.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/techproducts/update/extract?literal.id=doc6&defaultField
=text&commit=true" --data-binary @example/exampledocs/sample.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 SolrClient, 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 {
    SolrClient client = new
HttpSolrClient.Builder("http://localhost:8983/solr/my_collection").build();
    ContentStreamUpdateRequest req = new
ContentStreamUpdateRequest("/update/extract");
    req.addFile(new File("my-file.pdf"));
    req.setParam(ExtractingParams.EXTRACT_ONLY, "true");
    NamedList<Object> result = client.request(req);
    System.out.println("Result: " + result);
}
```

This operation streams the file `my-file.pdf` into the Solr index for `my_collection`.

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 `example/example-DIH` directory contains several collections many of the features of the data import handler. To run this "`dih`" example:

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

For more information about the Data Import Handler, see https://wiki.apache.org/solr/DataImportHandler.

Topics covered in this section:

- Concepts and Terminology
- Configuration
- Data Import Handler Commands
- Property Writer
- Data Sources
- Entity Processors
- Transformers
- Special Commands for the Data Import Handler

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

An annotated configuration file, based on the "`db`" collection in the `dih` example server, is shown below (`example/example-DIH/solr/db/conf/db-data-config.xml`). 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.
     The password attribute is optional if there is no password set for the DB.
-->
  <dataSource driver="org.hsqldb.jdbcDriver"
url="jdbc:hsqldb:./example-DIH/hsqldb/ex" user="sa" password="secret"/>
<!--
Alternately the password can be encrypted as follows. This is the value obtained as
a result of the command
openssl enc -aes-128-cbc -a -salt -in pwd.txt
password="U2FsdGVkX18QMjY0yfCqlfBMvAB4d3XkwY96L7gfO2o="
WHen the password is encrypted, you must provide an extra attribute
encryptKeyFile="/location/of/encryptionkey"
This file should a text file with a single line containing the encrypt/decrypt
password

-->
<!-- 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.

A `reload-config` command is also 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 and there is an interface to import content.

Request Parameters

Request parameters can be substituted in configuration with placeholder `${dataimporter.request.paramname}`.

```
<dataSource driver="org.hsqldb.jdbcDriver" url="${dataimporter.request.jdbcurl}"
user="${dataimporter.request.jdbcuser}"
password=${dataimporter.request.jdbcpassword} />
```

Then, these parameters can be passed to the full-import command or defined in the `<defaults>` section in `solrconfig.xml`. This example shows the parameters with the full-import command:

`dataimport?command=full-import&jdbcurl=jdbc:hsqldb:./example-DIH/hsqldb/ex&jdbcuser=sa&jdbcpassword=secret`

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/<collection_name>/dataimport?command=abort`.
`delta-import`	For incremental imports and change detection. The command is of the form `http://<host>:<port>/solr/<collection_name>/dataimport?command=delta-import`. It supports the same clean, commit, optimize and debug parameters as full-import command. Only the SqlEntityProcessor supports delta imports.

`full-import`	A Full Import operation can be started with a URL of the form `http://<host>:<port>/solr/<collection_name>/dataimport?command=full-import`. The command returns immediately. The operation will be started in a new thread and the *status* attribute in the response should be shown as *busy*. The operation may take some time depending on the size of dataset. Queries to Solr are not blocked during full-imports. When a full-import command is executed, it stores the start time of the operation in a file located at `conf/dataimport.properties`. This stored timestamp is used when a delta-import operation is executed. For a list of parameters that can be passed to this command, see below.
`reload-config`	If the configuration file has been changed and you wish to reload it without restarting Solr, run the command `http://<host>:<port>/solr/<collection_name>/command=reload-config`
`status`	The URL is `http://<host>:<port>/solr/<collection_name>/dataimport?command=status`. It returns statistics on the number of documents created, deleted, queries run, rows fetched, status, and so on.
`show-config`	responds with configuration

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.
synchronous	Blocks request until import is completed. Default is `false`.

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 (BCP 47 language tag). 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. If there are multiple data sources defined, use this attribute with the name of the data source for this entity.
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 "`SortedMapBackedCache`".
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.
where	an alternative way to specify `cacheKey` and `cacheLookup` concatenated with '='. eg `where="CODE=People.COUNTRY_CODE"` is equal to `cacheKey="CODE" cacheLookup="People.COUNTRY_CODE"`
child="true"	Enables indexing document blocks aka Nested Child Documents for searching with Block Join Query Parsers. It can be only specified on `<entity>` under another root entity. It switches from default behavior (merging field values) to nesting documents as children documents. Note: parent `<entity>` should add a field which is used as a parent filter in query time.
join="zipper"	Enables merge join aka "zipper" algorithm for joining parent and child entities without cache. It should be specified at child (nested) `<entity>`. It implies that parent and child queries return results ordered by keys, otherwise it throws an exception. Keys should be specified either with `where` attribute or with `cacheKey` and `cacheLookup`.

Caching of entities in DIH is provided to avoid repeated lookups for same entities again and again. The default `SortedMapBackedCache` 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.

Here is an example from the "`rss`" collection in the `dih` example (`example/example-DIH/solr/rss/conf/rss-data-config.xml`):

```
<!-- 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).

Here is an example from the "`mail`" collection of the `dih` example (`example/example-DIH/mail/conf/mail-data-config.xml`):

```
<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); excluded folder patterns take precedence over include folder patterns.
processAttachement or 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.

Here is an example from the "`tika`" collection of the `dih` example (`example/example-DIH/tika/conf/tika-data-config.xml`):

```
<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. The available data source types for this processor are: • BinURLDataSource: used for HTTP resources, but can also be used for files. • BinContentStreamDataSource: used for uploading content as a stream. • 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.
extractEmbedded	Instructs the TikaEntityProcessor to extract embedded documents or attachments when **true**. If false, embedded documents and attachments will be ignored.

onError	By default, the TikaEntityProcessor will stop processing documents if it finds one that generates an error. If you define `onError` to "skip", the TikaEntityProcessor will instead skip documents that fail processing and log a message that the document was skipped.

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 five implicit fields: `fileAbsolutePath, fileDir, fileSize, fileLastModified, file,` 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`).

SolrEntityProcessor

Uses Solr instance as a datasource, see https://wiki.apache.org/solr/DataImportHandler#SolrEntityProcessor

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 defined, the ROOT locale is used. It must be specified as language-country (BCP 47 language tag). 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. The locale. If not defined, the ROOT locale is used. It must be specified as language-country (BCP 47 language tag). 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. note: this is a regular expression – it may need to be escaped (e.g. via back-slashes)
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 8; 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 its 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. May be specified as a single value, or as a list for multivalued fields
add	Adds the specified values to a multivalued field. May be specified as a single value, or as a list.
remove	Removes (all occurrences of) the specified values from a multivalued field. May be specified as a single value, or as a list.
removeregex	Removes all occurrences of the specified regex from a multiValued field. May be specified as a single value, or as a list.
inc	Increments a numeric value by a specific amount. Must be specified as a single numeric value.

Field Storage

The core functionality of atomically updating a document requires that all fields in your schema must be configured as stored (`stored="true"`) except for fields which are `<copyField/>` destinations, which must be configured as `stored="false"`. Atomic updates are applied to the document represented by the existing stored field values. All data in these fields must originate from ONLY copyField sources.

If `<copyField/>` destinations are configured as stored, then Solr will attempt to index both the current value of the field as well as an additional copy from any source fields. If such fields contain some information that comes from the indexing program and some information that comes from copyField, then the information which originally came from the indexing program will likely be lost when an atomic update is made.

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":99,
 "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 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 the previous version.

When using Optimistic Concurrency, clients can include an optional `versions=true` request parameter to indicate that the *new* versions of the documents being added should be included in the response. This allows clients to immediately know what the `_version_` is of every documented added with out needing to make a redundant `/get` request.

For example...

```
$ curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/techproducts/update?versions=true' --data-binary '
[ { "id" : "aaa" },
  { "id" : "bbb" } ]'
{"responseHeader":{"status":0,"QTime":6},
 "adds":["aaa",1498562471222312960,
         "bbb",1498562471225458688]}
$ curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/techproducts/update?_version_=999999&versions=true'
--data-binary '
[{ "id" : "aaa",
   "foo_s" : "update attempt with wrong existing version" }]'
{"responseHeader":{"status":409,"QTime":3},
 "error":{"msg":"version conflict for aaa expected=999999
actual=1498562471222312960",
          "code":409}}
$ curl -X POST -H 'Content-Type: application/json'
'http://localhost:8983/solr/techproducts/update?_version_=1498562471222312960&versio
ns=true&commit=true' --data-binary '
[{ "id" : "aaa",
   "foo_s" : "update attempt with correct existing version" }]'
{"responseHeader":{"status":0,"QTime":5},
 "adds":["aaa",1498562624496861184]}
$ curl 'http://localhost:8983/solr/techproducts/query?q=*:*&fl=id,_version_'
{
  "responseHeader":{
    "status":0,
    "QTime":5,
    "params":{
      "fl":"id,_version_",
      "q":"*:*"}},
  "response":{"numFound":2,"start":0,"docs":[
      {
        "id":"bbb",
        "_version_":1498562471225458688},
      {
        "id":"aaa",
        "_version_":1498562624496861184}]
  }}
```

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 your custom `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.

versionField vs _version_
The `_version_` field used by Solr for its normal optimistic concurrency also has important semantics in how updates are distributed to replicas in SolrCloud, and **MUST** be assigned internally by Solr. Users can not re-purpose that field and specify it as the `versionField` for use in the `DocBasedVersionConstraintsProcessorFactory` configuration.

`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 DocBasedVersionConstraintsProcessorFactory javadocs and test solrconfig.xml file for additional information and example usages.

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: https://github.com/shuyo/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.LangDetectLanguageIdentifierUpdateProcessorF
actory">
  <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.

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. It's tunable but works best on longer text.

Other, more sophisticated algorithms for fuzzy/near hashing can be added later.



Adding in the de-duplication 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

There are two places in Solr to configure de-duplication: in `solrconfig.xml` and in `schema.xml`.

In `solrconfig.xml`

The `SignatureUpdateProcessorFactory` has to be registered in `solrconfig.xml` as part of an Update Request Processor Chain, as in this example:

```
<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>
   <processor class="solr.LogUpdateProcessorFactory" />
   <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

The `SignatureUpdateProcessorFactory` takes several properties:

Parameter	Default	Description

signatureClass	`org.apache.solr.update.processor.Lookup3Signature`	A Signature implementation for generating a signature hash. The full classpath of the implementation must be specified. The available options are described above, the associated classpaths to use are: • `org.apache.solr.update.processor.Lookup3Signature` • `org.apache.solr.update.processor.MD5Signature` • `org.apache.solr.update.process.TextProfileSignature`
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. The field should be defined in schema.xml.
enabled	true	Enable/disable de-duplication processing.
overwriteDupes	true	If true, when a document exists that already matches this signature, it will be overwritten.

In `schema.xml`

If you are using a separate field for storing the signature you must have it indexed:

```
<field name="signatureField" type="string" stored="true" indexed="true"
multiValued="false" />
```

Be sure to change your update handlers to use the defined chain, as below:

```
<requestHandler name="/update" class="solr.UpdateRequestHandler" >
  <lst name="defaults">
    <str name="update.chain">dedupe</str>
  </lst>
...
</requestHandler>
```

(This example assumes you have other sections of your request handler defined.)

The update processor can also be specified per request with a parameter of `update.chain=dedupe`.

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 convenience, remote streaming is enabled in most of the example `solrconfig.xml` files included with Solr, however it is not recommended in a production situation with out additional security between you and untrusted remote clients.

```
<!-- *** WARNING ***
         The settings below authorize Solr to fetch remote files, You
         should make sure your system has some authentication before
         using enableRemoteStreaming="true"
      -->
    <requestParsers enableRemoteStreaming="true" />
```

The default behavior, when `enableRemoteStreaming` is not specified in `solrconfig.xml` is to *not* allow remote streaming (i.e., `enableRemoteStreaming="false"`).



If you `enableRemoteStreaming="true"` is used, be aware that this allows *anyone* to send a request to any URL or local file. If DumpRequestHandler is enabled, it will allow anyone to view any file on your system.

Debugging Requests

The example `solrconfig.xml` files include 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-VERSION.jar` (under `/solr-VERSION/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<
/str>
      <!-- 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.chain">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 simple search UI 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 documents and multiple languages.
  - Postings Highlighter: Uses similar options as the FastVector highlighter, but is more compact and efficient.
- 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.
- 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.
- RealTime Get: How to get the latest version of a document without opening a searcher.
- Exporting Result Sets: Functionality to export large result sets out of Solr.
- Streaming Expressions: A stream processing language for Solr, with a suite of functions to perform many types of queries and parallel execution tasks.
- Parallel SQL Interface: An interface for sending SQL statements to Solr, and using advanced parallel query processing and relational algebra for complex data analysis.

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. Solr's default query parser is known as the Standard Query Parser,or more commonly just the "lucene" query parser. Solr also includes the DisMaxquery parser, and the 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 **filter query**. As part of a search response, a filter query 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. Filter queries 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 screen shot below shows an example of faceting from the CNET Web site (CBS Interactive Inc.) , which was the first site to use Solr.



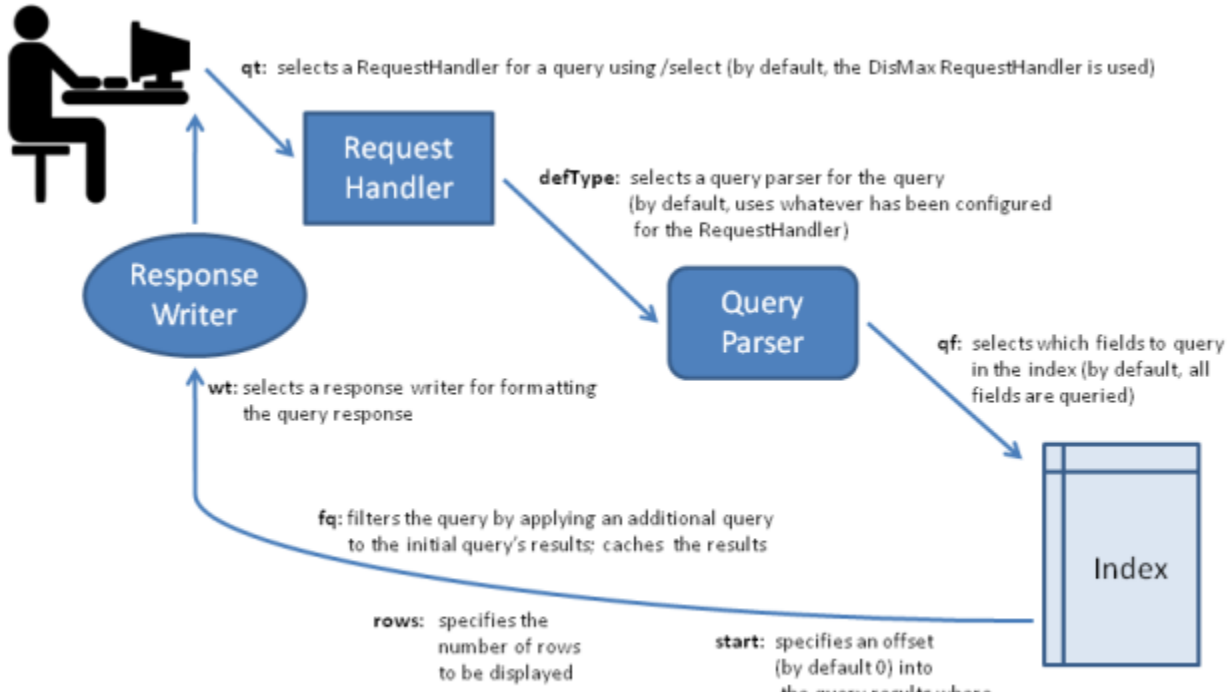
Faceting makes use of fields defined when the search applications were indexed. In the example above, these fields include categories of information that are useful for describing digital cameras: manufacturer, resolution, and zoom range.

Clustering groups search results by similarities discovered when a search is executed, rather than when content is indexed. The results of clustering often lack the neat hierarchical organization found in faceted search results, but clustering can be useful nonetheless. It can reveal unexpected commonalities among search results, and it can help users rule out content that isn't pertinent to what they're really searching for.

Solr also supports a feature called MoreLikeThis, which enables users to submit new queries that focus on particular terms returned in an earlier query. MoreLikeThis queries can make use of faceting or clustering to provide additional aid to users.

A Solr component called a **response writer** manages the final presentation of the query response. Solr includes a variety of response writers, including an XML Response Writer and a JSON Response Writer.

The diagram below summarizes some key elements of the search process.



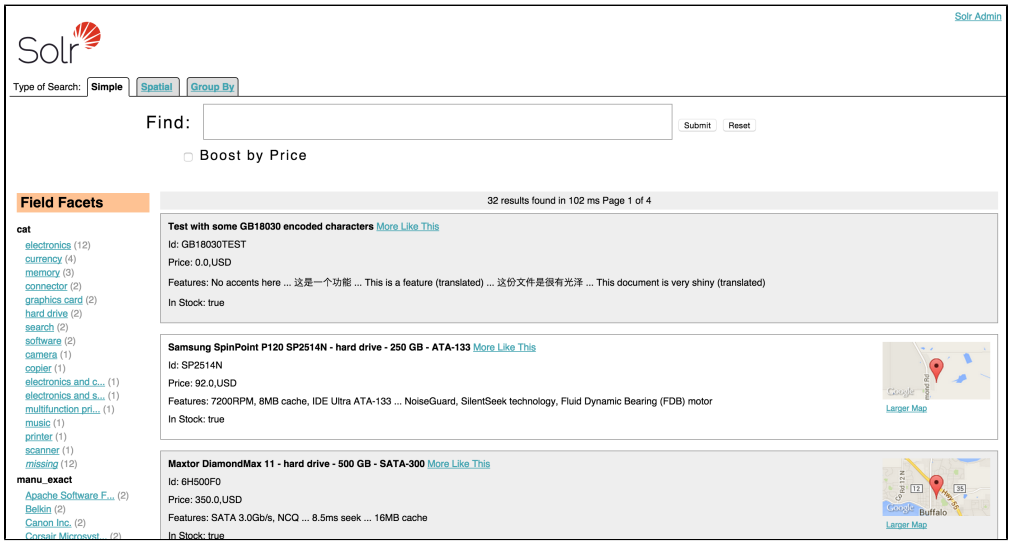
at one time

the query results where the returned response should begin

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.

When using the `sample_techproducts_configs` config set, you can access the Velocity sample Search UI here: `http://localhost:8983/solr/techproducts/browse`



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 te sts 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/SolrQuerySyntax.

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	Limits the information included in a query response to a specified list of fields. The fields need to either be `stored="true"` or `docValues="true"`
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.
segmentTerminateEarly	Indicates that, if possible, Solr should stop collecting documents from each individual (sorted) segment once it can determine that any subsequent documents in that segment will not be candidates for the `rows` being returned. The default is false.
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. 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.
echoParams	The response header can include parameters sent with the query request. This parameter controls what is contained in that section of the response header. Valid values are `none`, `all`, and `explicit`. The default value is `explicit`.

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`

If no defType param is specified, then by default, the The Standard Query Parser is used. (eg: `defType=lucene`)

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 field with a single value that is either indexed or uses DocValues (that is, any field whose attributes in the Schema include `multiValued="false"` and either `docValues="true"` or `indexed="true"` – if the field does not have DocValues enabled, the indexed terms are used to build them on the fly at runtime), 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 a `copyField` directive in the the Schema 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 either be `stored="true"` or `docValues="true".`

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 fields in the document which are either `stored="true"` or `docValues="true"` and `useDocValuesAsStored="true"` (which is the default when docValues are enabled). 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 `stored` fields in each document, as well as any `docValues` fields that have `useDocValuesAsStored="true"`. This is the default value of the fl parameter.
* score	Return all the fields in each document, along with each field's score.
*,dv_field_name	Return all the `stored` fields in each document, and any `docValues` fields that have `useDocValuesAsStored="true"` and the docValues from dv_field_name even if it has `useDocValuesAsStored="false"`

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

The `debug` parameter can be specified multiple times and supports the following arguments:

- `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 score results (also known as "explain").
  - By default, score explanations are returned as large string values, using newlines and tab indenting for structure & readability, but an additional `debug.explain.structured=true` parameter may be specified to return this information as nested data structures native to the response format requested by `wt`.
- `debug=all`: return all available debug information about the request request. (alternatively usage: `debug=true`)

For backwards compatibility with older versions of Solr, `debugQuery=true` may instead be specified as an alternative way to indicate `debug=all`

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, but values such as `numFound`, Facet c ounts, and result Stats may not be accurate for the entire result set.

The `segmentTerminateEarly` Parameter

This parameter may be set to either true or false.

If set to true, and if the `<mergePolicyFactory/>` for this collection is a `SortingMergePolicyFactory` whic h uses a `sort` option which is compatible with the `sort` parameter specified for this query, then Solr will attempt to use an `EarlyTerminatingSortingCollector`.

If early termination is used, a `segmentTerminatedEarly` header will be included in the `responseHeader`.

Similar to using the `timeAllowed` Parameter, when early segment termination happens values such as `numFo und`, Facet counts, and result Stats may not be accurate for the entire result set.

The default value of this parameter is false.

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 `echoParams` Parameter

The `echoParams` parameter controls what information about request parameters is included in the response header.

The table explains how Solr responds to various settings of the `echoParams` parameter:

Value	Meaning
explicit	This is the default value. Only parameters included in the actual request, plus the `_` parameter (which is a 64-bit numeric timestamp) will be added to the params section of the response header.
all	Include all request parameters that contributed to the query. This will include everything defined in the request handler definition found in `solrconfig.xml` as well as parameters included with the request, plus the `_` parameter. If a parameter is included in the request handler definition AND the request, it will appear multiple times in the response header.
none	Entirely removes the "params" section of the response header. No information about the request parameters will be available in the response.

Here is an example of a JSON response where the echoParams parameter was not included, so the default of `explicit` is active. The request URL that created this response included three parameters - `q`, `wt`, and `indent`:

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "params": {
      "q": "solr",
      "indent": "true",
      "wt": "json",
      "_": "1458227751857"
    }
  },
  "response": {
    "numFound": 0,
    "start": 0,
    "docs": []
  }
}
```

This is what happens if a similar request is sent that adds `echoParams=all` to the three parameters used in the previous example:

```
{
  "responseHeader": {
    "status": 0,
    "QTime": 0,
    "params": {
      "q": "solr",
      "df": "text",
      "preferLocalShards": "false",
      "indent": "true",
      "echoParams": "all",
      "rows": "10",
      "wt": "json",
      "_": "1458228887287"
    }
  },
  "response": {
    "numFound": 0,
    "start": 0,
    "docs": []
  }
}
```

The Standard Query Parser

Solr's default Query Parser is also known as the "`lucene`" parser.

The key advantage of the standard query parser is that it supports a robust and fairly intuitive syntax allowing you to create a variety of structured queries. The largest disadvantage is that it's very intolerant of syntax errors, as compared with something like the DisMax query parser which is designed to throw as few errors as possible.

Topics covered in this section:

- Standard Query Parser Parameters
- The Standard Query Parser's Response
- Specifying Terms for the Standard Query Parser
- Specifying Fields in a Query to the Standard Query Parser
- Boolean Operators Supported by the Standard Query Parser
- Grouping Terms to Form Sub-Queries
- Comments
- Differences between Lucene Query Parser and the Solr Standard Query Parser
- Related Topics

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. Possible values are "AND" or "OR".
df	Specifies a default field, overriding the definition of a default field in the Schema.

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://localhost:8983/solr/techproducts/select?q=id:SP2514N
```

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://localhost:8983/solr/techproducts/select?q=id:SP2514N&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:
		
		`tes*`
		
		would match test, testing, and tester.
		
		You can also use wildcard characters in the middle of a term. For example:
		
		`te*t`
		
		would match test and text.
		
		`*est`
		
		would match pest and test.

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).

Constant Score with ^=

Constant score queries are created with `<query_clause>^=<score>`, which sets the entire clause to the specified score for any documents matching that clause. This is desirable when you only care about matches for a particular clause and don't want other relevancy factors such as term frequency (the number of times the term appears in the field) or inverse document frequency (a measure across the whole index for how rare a term is in a field).

Example:

```
(description:blue OR color:blue)^=1.0 text:shoes
```

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. 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 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 functionally 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.

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")
```

Comments

C-Style comments are supported in query strings.

Example:

```
"jakarta apache" /* this is a comment in the middle of a normal query string */ OR jakarta
```

Comments may be nested.

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 using any type of query parser as a nested clause.
  - `inStock:true OR {!dismax qf='name manu' v='ipod'}`
- Support for a special `filter(...)` syntax to indicate that some query clauses should be cached in the filter cache (as a constant score boolean query). This allows sub-queries to be cached and re-used in other queries.
  For example `inStock:true` will be cached and re-used in all three of the queries below:
  - `q=features:songs OR filter(inStock:true)`
  - `q=+manu:Apple +filter(inStock:true)`
  - `q=+manu:Apple & fq=inStock:true`

  This can even be used to cache individual clauses of complex filter queries. In the first query below, 3 items will be added to the filter cache (the top level `fq` and both `filter(...)` clauses) and in the second query, there will be 2 cache hits, and one new cache insertion (for the new top level `fq`):

    - `q=features:songs & fq=+filter(inStock:true) +filter(price:[* TO 100])`
    - `q=manu:Apple & fq=-filter(inStock:true) -filter(price:[* TO 100])`
- 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 clauses 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 the Schema) 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. Default: 0.0
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-wide 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% 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.
- When searching across multiple fields that are configured with different query analyzers, the number of optional clauses may differ between the fields. In such a case, the value specified by mm applies to the maximum number of optional clauses. For example, if a query clause is treated as stopword for one of the fields, the number of optional clauses for that field will be smaller than for the other fields. A query with such a stopword clause would not return a match in that field if mm is set to 100% because the removed clause does not count as matched.

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" - the default - 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

All of the sample URLs in this section assume you are running Solr's "techproducts" example:

```
bin/solr -e techproducts
```

Normal results for the word "video" using the StandardRequestHandler with the default search field:

`http://localhost:8983/solr/techproducts/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/techproducts/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/techproducts/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/techproducts/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/techproducts/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/techproducts/select?defType=dismax&q=video&fl=name,score,inStock`

`http://localhost:8983/solr/techproducts/select?defType=dismax&q=video&qt=instoc`

```
k&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/techproducts/select?defType=dismax&q=belkin+ipod
```

```
http://localhost:8983/solr/techproducts/select?defType=dismax&q=belkin+ipod+gibberish
```

```
http://localhost:8983/solr/techproducts/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/techproducts/select?defType=dismax&q=belkin+ipod+gibberish&debugQuery=true
```

```
http://localhost:8983/solr/techproducts/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 the Schema, 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 mm.autoRelax Parameter

If true, the number of clauses required (minimum should match) will automatically be relaxed if a clause is removed (by e.g. stopwords filter) from some but not all `qf` fields. Use this parameter as a workaround if you experience that queries return zero hits due to uneven stopword removal between the `qf` fields.

Note that relaxing mm may cause undesired side effects, hurting the precision of the search, depending on the

nature of your index content.

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

This is similar to `ps` but overrides the slop factor used 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

This is similar to `ps` but overrides the slop factor used for `pf3`. If not specified, `ps` is 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

All of the sample URLs in this section assume you are running Solr's "`techproducts`" example:

```
bin/solr -e techproducts
```

Boost the result of the query term "hello" based on the document's popularity:

```
http://localhost:8983/solr/techproducts/select?defType=edismax&q=hello&pf=text&qf=te
xt&boost=popularity
```

Search for iPods OR video:

```
http://localhost:8983/solr/techproducts/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/techproducts/select?q=video&defType=edismax&qf=features^2
0.0+text^0.3
```

You can boost results that have a field that matches a specific value:

```
http://localhost:8983/solr/techproducts/select?q=video&defType=edismax&qf=features^2
0.0+text^0.3&bq=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/techproducts/select?q=belkin+ipod&defType=edismax&mm=2
http://localhost:8983/solr/techproducts/select?q=belkin+ipod+gibberish&defType=edism
ax&mm=2
http://localhost:8983/solr/techproducts/select?q=belkin+ipod+apple&defType=edismax&m
m=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:foo^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, in addition to the phrase fields (`pf`) parameter, `edismax` also supports the `pf2` and `pf3` parameters, for fields over which to create bigram and trigram phrase queries. The phrase slop for these parameters' queries can be specified using the `ps2` and `ps3` parameters, respectively. If you use `pf2`/`pf3` but `ps2`/`ps3`, then the phrase slop for these parameters' queries will be taken from the `ps` parameter, if any.

Using the 'magic fields' _val_ and _query_

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, 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
- Available Functions
- Example Function Queries
- Sort By Function
- Related Topics

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)` `abs(-5)`
"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 `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)` `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 `dist(1, x, y, 0, 0)`: calculates the Manhattan (taxicab) distance between (0,0) and (x,y) for each document `dist(2, x,y,z,0,0,0):` Euclidean distance between (0,0,0) and (x,y,z) for each document. `dist(1,x,y,z,e,f,g)`: Manhattan 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).
	
	You can quote the term if it's more complex, or do parameter substitution for the term value.
		`docfreq(text,'solr')`
		
		`...&defType=func &q=docfreq(text,$myterm) &myterm=solr`

field	Returns the numeric docValues or indexed value of the field with the specified name. In it's simplest (single argument) form, this function can only be used on single valued fields, and 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 with out using the `field(...)` syntax. When using docValues, an optional 2nd argument can be specified to select the "`min`" or "`max`" value of multivalued fields. 0 is returned for documents without a value in the field.	These 3 examples are all equivalent: • `myFloatFieldName` • `field(myFloatFieldName)` • `field("myFloatFieldName")` The last form is convinient when your field name is atypical: • `field("my complex float fieldName")` For multivalued docValues fields: • `field(myMultiValuedFloatField,min)` • `field(myMultiValuedFloatField,max)`
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)`: • `test` is or refers to a logical value or expression that returns a logical value (TRUE or FALSE). • `value1` is the value that is returned by the function if `test` yields TRUE. • `value2` is the value that is returned by the function if `test` yields FALSE. 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)`: 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)` `linear(x,2,4)` returns `2*x+4`
log	Returns the log base 10 of the specified function.	`log(x)` `log(sum(x,100))`

map	Maps any values of 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)`
		`map(x,0,0,1)` - changes any values of 0 to `1`. This can be useful in handling default 0 values.
		`map(x,min,max,target,default)`
		`map(x,0,100,1,-1)` - changes any values between `0` and `100` to `1`, and all other values to `-1`.
		`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 maximum numeric value of multiple nested functions or constants, which are specified as arguments: `max(x,y,...)`. The max function can also be useful for "bottoming out" another function or field at some specified constant. (Use the `field(myfield,max)` syntax for selecting the maximum value of a single multivalued field)	`max(myfield,myotherfield,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()`
min	Returns the minimum numeric value of multiple nested functions of constants, which are specified as arguments: `min(x,y,...)`. The min function can also be useful for providing an "upper bound" on a function using a constant. (Use the `field(myfield,min)` syntax for selecting the minimum value of a single multivalued field)	`min(myfield,myotherfield,0)`

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`.
	
	• `ms()`: Equivalent to `ms(NOW)`, number of milliseconds since the epoch.
	• `ms(a)`: Returns the number of milliseconds since the epoch that the argument represents.
	• `ms(a,b)` : Returns the number of milliseconds that b occurs before a (that is, a - b)
		`ms(NOW/DAY)`
		`ms(2000-01-01T00:00:00Z)`
		`ms(mydatefield)`
		`ms(NOW,mydatefield)`
		`ms(mydatefield, 2000-01-01T00:00:00Z)`
		
		`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)`

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()`
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. ⚠ `ord()` depends on the position in an index and can change when other documents are inserted or deleted. See also `rord` below.	`ord(myIndexedField)` 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)`
		`pow(x,log(y))`
		`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,...)`
		`product(x,2)`
		`product(x,y)`
		`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 `$otherparam` or direct specification of the query string in the Local Parameters through the `v` key.	`query(subquery, default)` `q=product(popularity, query({!dismax v='solr rocks'})`: returns the product of the popularity and the score of the DisMax query. `q=product(popularity, query($qq))&qq={!dismax}solr rocks`: equivalent to the previous query, using parameter de-referencing. `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(x,m,a,b)` implementing `a/(m*x+b)` where `m,a,b` are constants, and `x` is any arbitrarily complex function. When a and b are equal, and x>=0, this function has a maximum value of 1 that drops as x increases. Increasing the value of a and b together results in a movement of the entire function to a flatter part of the curve. These properties can make this an ideal function for boosting more recent documents when x is `rord(datefield)`.	`recip(myfield,m,a,b)` `recip(rord(creationDate),1,1000,1000)`
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. The current implementation cannot distinguish when documents have been deleted or documents that have no value. It uses 0.0 values for these cases. This means that if values are normally all greater than 0.0, one can still end up with 0.0 as the min value to map from. In these cases, an appropriate map() function could be used as a workaround to change 0.0 to a value in the real range, as shown here: scale(map(x,0,0,5),1,2)	`scale(x, minTarget, maxTarget)` `scale(x,1,2)`: scales the values of x such that all values will be between 1 and 2 inclusive.
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: jw: Jaro-Winkler edit: Levenstein or Edit distance ngram: The NGramDistance, if specified, can optionally pass in the ngram size too. Default is 2. 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)` `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)`
		`sum(x,y)`
		`sum(sqrt(x),log(y),z,0.5)`
		`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): `docFreq(fieldX:A)` = 2 (A appears in 2 docs) `freq(doc1, fieldX:A)` = 4 (A appears 4 times in doc 2) `totalTermFreq(fieldX:A)` = 5 (A appears 5 times across all docs) `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. 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')`

The following functions are boolean – they return true or false. They are mostly useful as the first argument of the `if` function, and some of these can be combined. If used somewhere else, it will yield a '1' or '0'.

Function	Description	Syntax Examples
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
or	A logical disjunction.	`or(value1,value2)`: TRUE if either `value1` or `value2` is true.
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.
not	The logically negated value of the wrapped function.	`not(exists(author))`: TRUE only when `exists(author)` is false.
exists	Returns TRUE if any member of the field exists.	`exists(author)` returns TRUE for any document has a value in the "author" field. `exists(query(price:5.00))` returns TRUE if "price" matches "5.00".
gt, gte, lt, lte, eq	5 comparison functions: Greater Than, Greater Than or Equal, Less Than, Less Than or Equal, Equal	`if(lt(ms(mydatefield),315569259747,0.8,1)` translates to this pseudocode: `if mydatefield < 315569259747 then 0.8 else 1`

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/collection_name/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/collection_name/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
```

If no "type" is specified (either explicitly or implicitly) then the lucene parser is used by default. Thus

```
fq={!df=summary}solr rocks
```

is equivilent to:

```
fq={!type=lucene df=summary}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
- Graph Query Parser
- Join Query Parser
- Lucene Query Parser
- Max Score Query Parser
- More Like This Query Parser
- Nested Query Parser
- Old Lucene Query Parser
- Prefix Query Parser
- Raw Query Parser
- Re-Ranking Query Parser
- Simple Query Parser
- Spatial Query Parsers
- Surround Query Parser
- Switch Query Parser
- Term Query Parser
- Terms Query Parser
- XML 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 has 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">New Lucene and Solr release</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 **all parent documents**. The parameter `someParents` identifies a query that will match some 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 **all parent documents**. 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 has block join support</str></arr>
    <arr name="content_type"><str>parentDocument</str></arr>
  </doc>
</result>
```

Using which

A common mistake is to try to filter parents with a `which` filter, as in this bad example:

```
q={!parent which="title:join"}comments:SolrCloud
```

Instead, you should use a sibling mandatory clause as a filter:

```
q= +title:join +{!parent which="content_type:parentDocument"}comments:SolrCloud
```

Scoring

You can optionally use the `score` local parameter to return scores of the subordinate query. The values to use for this parameter define the type of aggregation, which are `avg` (average), `max` (maximum), `min` (minimum), `total (sum)`. Implicit default is `none` which returns `0.0`.

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 section Collapse and Expand Results.

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.

Graph Query Parser

The `graph` query parser does a breadth first, cyclic aware, graph traversal of all documents that are "reachable" from a starting set of root documents identified by a wrapped query. The graph is built according to linkages between documents based on the terms found in "`from`" and "`to`" fields that you specify as part of the query

Parameters

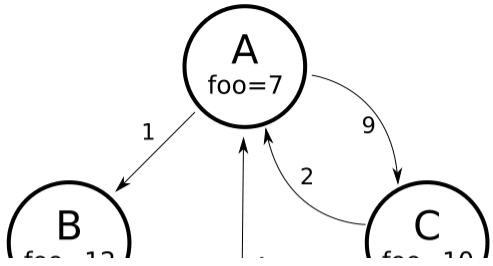
Parameter	Description
to	The field name of matching documents to inspect to identify outgoing edges for graph traversal. Defaults to `edge_ids` .
from	The field name to of candidate documents to inspect to identify incoming graph edges. Defaults to `node_id` .
traversalFilter	An optional query that can be supplied to limit the scope of documents that are traversed.
maxDepth	Integer specifying how deep the breadth first search of the graph should go begining with the initial query. Defaults to -1 (unlimited)
returnRoot	Boolean to indicate if the documents that matched the original query (to define the starting points for graph) should be included in the final results. Defaults to true
returnOnlyLeaf	Boolean that indicates if the results of the query should be filtered so that only documents with no outgoing edges are returned. Defaults to false
useAutn	Boolean that indicates if an Automatons should be compiled for each iteration of the breadth first search, which may be faster for some graphs. Defaults to false.

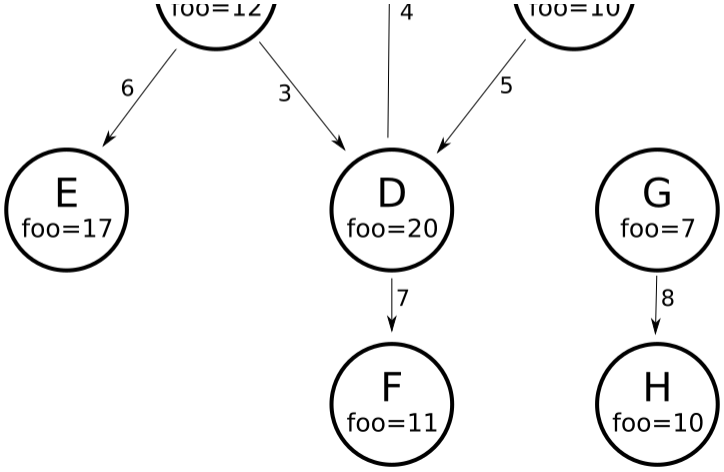
Limitations

The `graph` parser only works in single node Solr installations, or with SolrCloud collections that use exactly 1 shard.

Examples

To understand how the graph parser works, consider the following Directed Cyclic Graph, containing 8 nodes (A to H) and 9 edges (1 to 9):





One way to model this graph as Solr documents, would be to create one document per node, with mutivalued fields identifying the incoming and outgoing edges for each node:

```
curl -H 'Content-Type: application/json'
'http://localhost:8983/solr/my_graph/update?commit=true' --data-binary '[
  {"id":"A","foo":  7, "out_edge":["1","9"],  "in_edge":["4","2"]  },
  {"id":"B","foo": 12, "out_edge":["3","6"],  "in_edge":["1"]      },
  {"id":"C","foo": 10, "out_edge":["5","2"],  "in_edge":["9"]      },
  {"id":"D","foo": 20, "out_edge":["4","7"],  "in_edge":["3","5"]  },
  {"id":"E","foo": 17, "out_edge":[],         "in_edge":["6"]      },
  {"id":"F","foo": 11, "out_edge":[],         "in_edge":["7"]      },
  {"id":"G","foo":  7, "out_edge":["8"],      "in_edge":[]         },
  {"id":"H","foo": 10, "out_edge":[],         "in_edge":["8"]      }
]'
```

With the model shown above, the following query demonstrates a simple traversal of all nodes reachable from node A:

```
http://localhost:8983/solr/my_graph/query?fl=id&q={!graph+from=in_edge+to=out_edge}i
d:A
...
"response":{"numFound":6,"start":0,"docs":[
   { "id":"A" },
   { "id":"B" },
   { "id":"C" },
   { "id":"D" },
   { "id":"E" },
   { "id":"F" } ]
}
```

We can also use the `traversalFilter` to limit the graph traversal to only nodes with maximum value of 15 in the `foo` field. In this case that means D, E, and F are excluded – F has a value of `foo=11`, but it is unreachable because the traversal skipped D:

```
http://localhost:8983/solr/my_graph/query?fl=id&q={!graph+from=in_edge+to=out_edge+t
raversalFilter='foo:[*+TO+15]'}id:A
...
"response":{"numFound":3,"start":0,"docs":[
   { "id":"A" },
   { "id":"B" },
   { "id":"C" } ]
}
```

The examples shown so far have all used a query for a single document (`"id:A"`) as the root node for the graph traversal, but any query can be used to identify multiple documents to use as root nodes. The next example demonstrates using the `maxDepth` param to find all nodes that are at most one edge away from an root node with a value in the `foo` field less then or equal to 10:

```
http://localhost:8983/solr/my_graph/query?fl=id&q={!graph+from=in_edge+to=out_edge+m
axDepth=1}foo:[*+TO+10]
...
"response":{"numFound":6,"start":0,"docs":[
   { "id":"A" },
   { "id":"B" },
   { "id":"C" },
   { "id":"D" },
   { "id":"G" },
   { "id":"H" } ]
}
```

Simplified Models

The Document & Field modelling used in the above examples enumerated all of the outgoing and income edges for each node explicitly, to help demonstrate exactly how the "from" and "to" params work, and to give you an idea of what is possible. With multiple sets of fields like these for identifying incoming and outgoing edges, it's possible to model many independent Directed Graphs that contain some or all of the documents in your collection.

But in many cases it can also be possible to drastically simplify the model used.

For Example: The same graph shown in the diagram above can be modelled by Solr Documents that represent each node and know only the ids of the nodes they link to, with out knowing anything about the incoming links:

```
curl -H 'Content-Type: application/json'
'http://localhost:8983/solr/alt_graph/update?commit=true' --data-binary '[
  {"id":"A","foo":  7, "out_edge":["B","C"] },
  {"id":"B","foo": 12, "out_edge":["E","D"] },
  {"id":"C","foo": 10, "out_edge":["A","D"] },
  {"id":"D","foo": 20, "out_edge":["A","F"] },
  {"id":"E","foo": 17, "out_edge":[]        },
  {"id":"F","foo": 11, "out_edge":[]        },
  {"id":"G","foo":  7, "out_edge":["H"]     },
  {"id":"H","foo": 10, "out_edge":[]        }
  ]'
```

With this alternative document model, all of the same queries demonstrated above can still be executed, simply by changing the "`from`" param to replace the "`in_edge`" field with the "`id`" field:

```
http://localhost:8983/solr/alt_graph/query?fl=id&q={!graph+from=id+to=out_edge+maxDe
pth=1}foo:[*+TO+10]
...
"response":{"numFound":6,"start":0,"docs":[
   { "id":"A" },
   { "id":"B" },
   { "id":"C" },
   { "id":"D" },
   { "id":"G" },
   { "id":"H" } ]
}
```

Join Query Parser

`JoinQParser` extends the `QParserPlugin`. It allows normalizing relationships between documents with a join operation. This is different from the 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]
```

The join operation is done on a term basis, so the "from" and "to" fields must use compatible field types. For example: joining between a `StrField` and a `TrieIntField` will not work, likewise joining between a `StrField` and a `TextField` that uses `LowerCaseFilterFactory` will only work for values that are already lower cased in the string field.

Scoring

You can optionally use the `score` parameter to return scores of the subordinate query. The values to use for this parameter define the type of aggregation, which are `avg` (average), `max` (maximum), `min` (minimum) `total`, or `none`.

Joining Across Collections

You can also specify a `fromIndex` parameter to join with a field from another core or collection. If running in SolrCloud mode, then the collection specified in the `fromIndex` parameter must have a single shard and a replica on all Solr nodes where the collection you're joining to has a replica.

Let's consider an example where you want to use a Solr join query to filter movies by directors that have won an Oscar. Specifically, imagine we have two collections with the following fields:

movies: id, title, director_id, ...

movie_directors: id, name, has_oscar, ...

To filter movies by directors that have won an Oscar using a Solr join on the **movie_directors** collection, you can send the following filter query to the **movies** collection:

```
fq={!join from=id fromIndex=movie_directors to=director_id}has_oscar:true
```

Notice that the query criteria of the filter (`has_oscar:true`) is based on a field in the collection specified using `fromIndex`. Keep in mind that you cannot return fields from the `fromIndex` collection using join queries, you can only use the fields for filtering results in the "to" collection (movies).

Next, let's understand how these collections need to be deployed in your cluster. Imagine the **movies** collection is deployed to a four node SolrCloud cluster and has two shards with a replication factor of two. Specifically, the **movies** collection has replicas on the following four nodes:

node 1: movies_shard1_replica1

node 2: movies_shard1_replica2

node 3: movies_shard2_replica1

node 4: movies_shard2_replica2

To use the **movie_directors** collection in Solr join queries with the **movies** collection, it needs to have a replica on each of the four nodes. In other words, **movie_directors** must have one shard and replication factor of four:

node 1: movie_directors_shard1_replica1

node 2: movie_directors_shard1_replica2

node 3: movie_directors_shard1_replica3

node 4: movie_directors_shard1_replica4

At query time, the `JoinQParser` will access the local replica of the **movie_directors** collection to perform the join. If a local replica is not available or active, then the query will fail. At this point, it should be clear that since you're limited to a single shard and the data must be replicated across all nodes where it is needed, this approach works better with smaller data sets where there is a one-to-many relationship between the from collection and the to collection. Moreover, if you add a replica to the to collection, then you also need to add a replica for the from collection.

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)
```

More Like This Query Parser

`MLTQParser` enables retrieving documents that are similar to a given document. It uses Lucene's existing `More LikeThis` logic and also works in SolrCloud mode. The document identifier used here is the unique id value and not the Lucene internal document id. The list of returned documents excludes the queried document.

This query parser takes the following parameters:

Parameter	Description
qf	Specifies the fields to use for similarity.
mintf	Specifies the Minimum Term Frequency, the frequency below which terms will be ignored in the source document.
mindf	Specifies the Minimum Document Frequency, the frequency at which words will be ignored when they do not occur in at least this many documents.
maxdf	Specifies the Maximum Document Frequency, the frequency at which words will be ignored when they occur in more than this many documents.
minwl	Sets the minimum word length below which words will be ignored.
maxwl	Sets the maximum word length above which words will be ignored.
maxqt	Sets the maximum number of query terms that will be included in any generated query.
maxntp	Sets the maximum number of tokens to parse in each example document field that is not stored with TermVector support.
boost	Specifies if the query will be boosted by the interesting term relevance. It can be either "true" or "false".

Examples:

Find documents like the document with id=1 and using the `name` field for similarity.

```
{!mlt qf=name}1
```

Adding more constraints to what qualifies as similar using mintf and mindf.

```
{!mlt qf=name mintf=2 mindf=3}1
```

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 Query Re-Ranking 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.operators</td><td>Comma-separated list of names of parsing operators to enable. By default, all operations are enabled, and this parameter can be used to effectively disable specific operators as needed, by excluding them from the list. Passing an empty string with this parameter disables all operators.
<table>
<tr><th>Name</th><th>Operator</th><th>Description</th><th>Example query</th></tr>
<tr><td>AND</td><td>+</td><td>Specifies AND</td><td>token1+token2</td></tr>
<tr><td>OR</td><td>|</td><td>Specifies OR</td><td>token1|token2</td></tr>
<tr><td>NOT</td><td>-</td><td>Specifies NOT</td><td>-token3</td></tr>
<tr><td>PREFIX</td><td>*</td><td>Specifies a prefix query</td><td>term*</td></tr>
<tr><td>PHRASE</td><td>"</td><td>Creates a phrase</td><td>"term1 term2"</td></tr>
<tr><td>PRECEDENCE</td><td>( )</td><td>Specifies precedence; tokens inside the parenthesis will be analyzed first. Otherwise, normal order is left to right.</td><td>token1 + (token2 | token3)</td></tr>
<tr><td>ESCAPE</td><td>\</td><td>Put it in front of operators to match them literally</td><td>C\+\+</td></tr>
<tr><td>WHITESPACE</td><td>space or [\r\t\n]</td><td>Delimits tokens on whitespace. If not enabled, whitespace splitting will not be performed prior to analysis – usually most desirable. Not splitting whitespace is a unique feature of this parser that enables multi-word synonyms to work. However, it probably actually won't unless synonyms are configured to normalize instead of expand to all that match a given synonym. Such a configuration requires normalizing synonyms at both index time and query time. Solr's analysis screen can help here.</td><td>term1 term2</td></tr>
<tr><td>FUZZY</td><td>~N</td><td>At the end of terms, specifies a fuzzy query</td><td>term~1</td></tr>
<tr><td>NEAR</td><td>~N</td><td>At the end of phrases, specifies a NEAR query</td><td>"term1 term2"~5</td></tr>
</table>
</td></tr>
</table>

q.op	Defines the default operator to use if none is defined by the user. Allowed values are `AND` and `OR`. `OR` is used if none is specified.
qf	A list of query fields and boosts to use when building the query.
df	Defines the default field if none is defined in the Schema, or overrides the default field if it is already defined.

Any errors in syntax are ignored and the query parser will interpret queries as best it can. However, this can lead to odd results in some cases.

Spatial Query Parsers

There are two spatial QParsers in Solr: `geofilt` and `bbox`. But there are other ways to query spatially: using the `frange` parser with a distance function, using the standard (lucene) query parser with the range syntax to pick the corners of a rectangle, or with RPT and BBoxField you can use the standard query parser but use a special syntax within quotes that allows you to pick the spatial predicate.

All these things are documented further in the section Spatial Search.

Surround Query Parser

The `SurroundQParser` enables the Surround query syntax, which provides proximity search functionality. There are two positional 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.

The non-unary operators (everything but `NOT`) support both infix `(a AND b AND c)` and prefix `AND(a, b, c)` notation.

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 than 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 query-building implementation: `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
```

XML Query Parser

The XmlQParserPlugin extends the QParserPlugin and supports the creation of queries from XML. Example:

Parameter	Value
defType	`xmlparser`

q	`<BooleanQuery fieldName="description">`
	`<Clause occurs="must"> <TermQuery>shirt</TermQuery> </Clause>`
	`<Clause occurs="mustnot"> <TermQuery>plain</TermQuery> </Clause>`
	`<Clause occurs="should"> <TermQuery>cotton</TermQuery> </Clause>`
	`<Clause occurs="must">`
	`<BooleanQuery fieldName="size">`
	`<Clause occurs="should"> <TermsQuery>S M L</TermsQuery> </Clause>`
	`</BooleanQuery>`
	`</Clause>`
	`</BooleanQuery>`

The XmlQParser implementation uses the SolrCoreParser class which extends Lucene's CoreParser class. XML elements are mapped to QueryBuilder classes as follows:

XML element	QueryBuilder class
<BooleanQuery>	BooleanQueryBuilder
<BoostingTermQuery>	BoostingTermBuilder
<ConstantScoreQuery>	ConstantScoreQueryBuilder
<DisjunctionMaxQuery>	DisjunctionMaxQueryBuilder

<MatchAllDocsQuery>	MatchAllDocsQueryBuilder
<RangeQuery>	RangeQueryBuilder
<SpanFirst>	SpanFirstBuilder
<SpanNear>	SpanNearBuilder
<SpanNot>	SpanNotBuilder
<SpanOr>	SpanOrBuilder
<SpanOrTerms>	SpanOrTermsBuilder
<SpanTerm>	SpanTermBuilder
<TermQuery>	TermQueryBuilder
<TermsQuery>	TermsQueryBuilder
<UserQuery>	UserInputQueryBuilder
<LegacyNumericRangeQuery>	LegacyNumericRangeQuery(Builder) is deprecated

Customizing XML Query Parser

You can configure your own custom query builders for additional XML elements. The custom builders need to extend the SolrQueryBuilder class. Example solrconfig.xml snippet:

```
<queryParser name="xmlparser" class="XmlQParserPlugin">
  <str name="MyCustomQuery">com.mycompany.solr.search.MyCustomQueryBuilder</str>
</queryParser>
```

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
- Field-Value Faceting Parameters
- Range Faceting
- Pivot (Decision Tree) Faceting
- Interval Faceting
- Local Parameters for Faceting
- Related Topics

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 your Schema 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.contains	Limits the terms used for faceting to those that contain the specified substring.
facet.contains.ignoreCase	If facet.contains is used, ignore case when searching for the specified substring.
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.contains` Parameter

The `facet.contains` parameter limits the terms on which to facet to those containing the given substring. 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.contains`.

The `facet.contains.ignoreCase` Parameter

If `facet.contains` is used, the `facet.contains.ignoreCase` parameter causes case to be ignored

when matching the given substring against candidate facet terms.

This parameter can be specified on a per-field basis with the syntax of `f.<fieldname>.facet.contains.ignoreCase`.

The `facet.sort` Parameter

This parameter determines the ordering of the facet field constraints.

`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.
facet.range.method	Specifies the algorithm or method to use for calculating facets.

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
```

```
facet.range=lastModified_dt
```

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
```

```
f.lastModified_dt.facet.range.start=NOW/DAY-30DAYS
```

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
```

```
f.lastModified_dt.facet.range.end=NOW/DAY+30DAYS
```

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
```

```
f.lastModified_dt.facet.range.gap=+1DAY
```

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.range.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.

The `facet.range.method` Parameter

The `facet.range.method` parameter selects the type of algorithm or method Solr should use for range faceting. Both methods produce the same results, but performance may vary.

Method	Description
filter	This method generates the ranges based on other facet.range parameters, and for each of them executes a filter that later intersects with the main query resultset to get the count. It will make use of the filterCache, so it will benefit of a cache large enough to contain all ranges.
dv	This method iterates the documents that match the main query, and for each of them finds the correct range for the value. This method will make use of docValues (if enabled for the field) or fieldCache. "dv" method is not supported for field type DateRangeField or when using group.facets.

Default value for this parameter is "filter".

The facet.mincount Parameter in Range Faceting

The `facet.mincount` parameter, the same one as used in field faceting is also applied to range faceting. When used, no ranges with a count below the minimum will be included in the response.

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.

Pivot (Decision Tree) Faceting

Pivoting is a summarization tool that lets you automatically sort, count, total or average data stored in a table. The results are typically displayed in a second table showing the summarized data. Pivot faceting lets you create a summary table of the results from a faceting documents by multiple 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.

Using the "`bin/solr -e techproducts`" example, A query URL like this one will returns the data below, with the pivot faceting results found in the section "facet_pivot":

```
http://localhost:8983/solr/techproducts/select?q=*:*&facet.pivot=cat,popularity,i
nStock
   &facet.pivot=popularity,cat&facet=true&facet.field=cat&facet.limit=5
   &rows=0&wt=json&indent=true&facet.pivot.mincount=2
```

```
"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}]},
...
```

Combining Stats Component With Pivots

In addition to some of the general local parameters supported by other types of faceting, a `stats` local parameters can be used with `facet.pivot` to refer to `stats.field` instances (by tag) that you would like to have computed for each Pivot Constraint.

In the example below, two different (overlapping) sets of statistics are computed for each of the facet.pivot result hierarchies:

```
stats=true
stats.field={!tag=piv1,piv2 min=true max=true}price
stats.field={!tag=piv2 mean=true}popularity
facet=true
facet.pivot={!stats=piv1}cat,inStock
facet.pivot={!stats=piv2}manu,inStock
```

Results:

```
"facet_pivot":{
  "cat,inStock":[{
      "field":"cat",
      "value":"electronics",
      "count":12,
      "pivot":[{
          "field":"inStock",
          "value":true,
          "count":8,
          "stats":{
            "stats_fields":{
```

```
            "price":{
              "min":74.98999786376953,
              "max":399.0}}}},
       {
         "field":"inStock",
         "value":false,
         "count":4,
         "stats":{
           "stats_fields":{
             "price":{
               "min":11.5,
               "max":649.989990234375}}}}],
     "stats":{
       "stats_fields":{
         "price":{
           "min":11.5,
           "max":649.989990234375}}}},
   {
     "field":"cat",
     "value":"currency",
     "count":4,
     "pivot":[{
         "field":"inStock",
         "value":true,
         "count":4,
         "stats":{
           "stats_fields":{
             "price":{
               ...
 "manu,inStock":[{
     "field":"manu",
     "value":"inc",
     "count":8,
     "pivot":[{
         "field":"inStock",
         "value":true,
         "count":7,
         "stats":{
           "stats_fields":{
             "price":{
               "min":74.98999786376953,
               "max":2199.0},
             "popularity":{
               "mean":5.857142857142857}}}},
       {
         "field":"inStock",
         "value":false,
         "count":1,
         "stats":{
           "stats_fields":{
             "price":{
               "min":479.95001220703125,
               "max":479.95001220703125},
             "popularity":{
```

```
                "mean":7.0}}}}],
      ...
```

Combining Facet Queries And Facet Ranges With Pivot Facets

A `query` local parameter can be used with `facet.pivot` to refer to `facet.query` instances (by tag) that should be computed for each pivot constraint. Similarly, a `range` local parameter can be used with `facet.pivot` to refer to `facet.range` instances.

In the example below, two query facets are computed for each of the `facet.pivot` result hierarchies:

```
facet=true
facet.query={!tag=q1}manufacturedate_dt:[2006-01-01T00:00:00Z TO NOW]
facet.query={!tag=q1}price:[0 TO 100]
facet.pivot={!query=q1}cat,inStock
```

```
"facet_counts": {
    "facet_queries": {
      "{!tag=q1}manufacturedate_dt:[2006-01-01T00:00:00Z TO NOW]": 9,
      "{!tag=q1}price:[0 TO 100]": 7
    },
    "facet_fields": {},
    "facet_dates": {},
    "facet_ranges": {},
    "facet_intervals": {},
    "facet_heatmaps": {},
    "facet_pivot": {
      "cat,inStock": [
        {
          "field": "cat",
          "value": "electronics",
          "count": 12,
          "queries": {
            "{!tag=q1}manufacturedate_dt:[2006-01-01T00:00:00Z TO NOW]": 9,
            "{!tag=q1}price:[0 TO 100]": 4
          },
          "pivot": [
            {
              "field": "inStock",
              "value": true,
              "count": 8,
              "queries": {
                "{!tag=q1}manufacturedate_dt:[2006-01-01T00:00:00Z TO NOW]": 6,
                "{!tag=q1}price:[0 TO 100]": 2
              }
            },
   ...
```

In a similar way, in the example below, two range facets are computed for each of the `facet.pivot` result hierarchies:

```
facet=true
facet.range={!tag=r1}manufacturedate_dt
facet.range.start=2006-01-01T00:00:00Z
facet.range.end=NOW/YEAR
facet.range.gap=+1YEAR
facet.pivot={!range=r1}cat,inStock
```

```
"facet_counts":{
    "facet_queries":{},
    "facet_fields":{},
    "facet_dates":{},
    "facet_ranges":{
      "manufacturedate_dt":{
        "counts":[
          "2006-01-01T00:00:00Z",9,
          "2007-01-01T00:00:00Z",0,
          "2008-01-01T00:00:00Z",0,
          "2009-01-01T00:00:00Z",0,
          "2010-01-01T00:00:00Z",0,
          "2011-01-01T00:00:00Z",0,
          "2012-01-01T00:00:00Z",0,
          "2013-01-01T00:00:00Z",0,
          "2014-01-01T00:00:00Z",0],
        "gap":"+1YEAR",
        "start":"2006-01-01T00:00:00Z",
        "end":"2015-01-01T00:00:00Z"}},
    "facet_intervals":{},
    "facet_heatmaps":{},
    "facet_pivot":{
      "cat,inStock":[{
          "field":"cat",
          "value":"electronics",
          "count":12,
          "ranges":{
            "manufacturedate_dt":{
              "counts":[
                "2006-01-01T00:00:00Z",9,
                "2007-01-01T00:00:00Z",0,
                "2008-01-01T00:00:00Z",0,
                "2009-01-01T00:00:00Z",0,
                "2010-01-01T00:00:00Z",0,
                "2011-01-01T00:00:00Z",0,
                "2012-01-01T00:00:00Z",0,
                "2013-01-01T00:00:00Z",0,
                "2014-01-01T00:00:00Z",0],
              "gap":"+1YEAR",
              "start":"2006-01-01T00:00:00Z",
              "end":"2015-01-01T00:00:00Z"}},
          "pivot":[{
              "field":"inStock",
              "value":true,
              "count":8,
              "ranges":{
                "manufacturedate_dt":{
                  "counts":[
                    "2006-01-01T00:00:00Z",6,
```

```
                    "2007-01-01T00:00:00Z",0,
                    "2008-01-01T00:00:00Z",0,
                    "2009-01-01T00:00:00Z",0,
                    "2010-01-01T00:00:00Z",0,
                    "2011-01-01T00:00:00Z",0,
                    "2012-01-01T00:00:00Z",0,
                    "2013-01-01T00:00:00Z",0,
                    "2014-01-01T00:00:00Z",0],
                  "gap":"+1YEAR",
```

```
                  "start":"2006-01-01T00:00:00Z",
                  "end":"2015-01-01T00:00:00Z"}}},
      ...
```

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.

Even though the same functionality can be achieved by using a 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 filter cache is more effective (static indexes for example). This method will use docValues if they are enabled for the field, will use fieldCache otherwise.

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.

```
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 a closing ')' 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 cannot 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 but 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. The 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].

Interval faceting supports output key replacement described below. Output keys can be replaced in both the `facet.interval parameter` and in the `facet.interval.set parameter`. For example:

```
&facet.interval={!key=popularity}some_field
&facet.interval.set={!key=bad}[0,5]
&facet.interval.set={!key=good}[5,*]
&facet=true
```

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.

Limiting facet with certain terms

To limit field facet with certain terms specify them comma separated with `terms` local parameter. Commas and quotes in terms can be escaped with backslash \,. In this case facet is calculated on a way similar to `facet.method=enum`, but ignores `facet.enum.cache.minDf`. For example:

```
facet.field={!terms='alfa,betta,with\,with\',with space'}symbol
```

Related Topics

- SimpleFacetParameters from the Solr Wiki.
- Heatmap Faceting (Spatial)

BlockJoin Faceting

It's a common requirement to aggregate children facet counts by their parents, i.e., if a parent document has several children documents, all of them need to increment facet value count only once. This functionality is provided by `BlockJoinDocSetFacetComponent`, and `BlockJoinFacetComponent` just an alias for compatibility.

This component is considered experimental, and must be explicitly enabled for a request handler in `solrconfig.xml`, in the same way as any other search component.

This example shows how you could add this search components to `solrconfig.xml` and define it in request handler:

solrconfig.xml

```
<searchComponent name="bjqFacetComponent"
class="org.apache.solr.search.join.BlockJoinDocSetFacetComponent"/>

   <requestHandler name="/bjqfacet"
class="org.apache.solr.handler.component.SearchHandler">
    <lst name="defaults">
      <str name="shards.qt">/bjqfacet</str>
    </lst>
    <arr name="last-components">
      <str>bjqFacetComponent</str>
    </arr>
  </requestHandler>
```

This component can be added into any search request handler. This component work with distributed search in SolrCloud mode.

Documents should be added in children-parent blocks as described in indexing nested child documents. Examples:

document sample

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="type_s">parent</field>
    <doc>
      <field name="id">11</field>
      <field name="COLOR_s">Red</field>
      <field name="SIZE_s">XL</field>
      <field name="PRICE_i">6</field>
    </doc>
    <doc>
      <field name="id">12</field>
      <field name="COLOR_s">Red</field>
      <field name="SIZE_s">XL</field>
      <field name="PRICE_i">7</field>
    </doc>
    <doc>
      <field name="id">13</field>
      <field name="COLOR_s">Blue</field>
      <field name="SIZE_s">L</field>
      <field name="PRICE_i">5</field>
    </doc>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="type_s">parent</field>
    <doc>
      <field name="id">21</field>
      <field name="COLOR_s">Blue</field>
      <field name="SIZE_s">XL</field>
      <field name="PRICE_i">6</field>
    </doc>
    <doc>
      <field name="id">22</field>
      <field name="COLOR_s">Blue</field>
      <field name="SIZE_s">XL</field>
      <field name="PRICE_i">7</field>
    </doc>
    <doc>
      <field name="id">23</field>
      <field name="COLOR_s">Red</field>
      <field name="SIZE_s">L</field>
      <field name="PRICE_i">5</field>
    </doc>
  </doc>
</add>
```

Queries are constructed the same way as for a Parent Block Join query. For example:

```
http://localhost:8983/solr/bjqfacet?q={!parent
which=type_s:parent}SIZE_s:XL&child.facet.field=COLOR_s
```

As a result we should have facets for Red(1) and Blue(1), because matches on children `id=11` and `id=12` are

aggregated into single hit into parent with `id=1`. The key components of the request are:

url part	meaning
`/bjqfacet`	The name of the request handler that has been defined with one of block join facet components enabled.
`q={!parent ...}..`	The mandatory parent query as a main query. The parent query could also be a subordinate clause in a more complex query.
`child.facet.field=...`	The child document field, which might be repeated many times with several fields, as necessary.

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 `solrconfig.xml` used in the "`techproducts`" example which shows 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 (AKA the default highlighter) doesn't require any special indexing parameters on the fields to highlight. However you can optionally turn on `termVectors`, `termPositions`, and `termOffsets` for any field to be highlighted. This will avoid having to run documents through the analysis chain at query-time and will make highlighting significantly faster and use less memory, particularly for large text fields, and even more so when `hl.usePhraseHighlighter` is enabled.

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. 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.highlightAlternate	true	If set to **true**, and `hl.alternateFieldName` is active, Solr will show the entire alternate field, with highlighting of occurrences. If `hl.maxAlternateFieldLength=N` is used, Solr returns max `N` characters surrounding the best matching fragment. If set to **false**, or if there is no match in the alternate field either, the alternate field will be shown without highlighting.
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 hl.simple.post	and	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 highlight phrase queries (and other advanced position-sensitive queries) accurately. If **false**, the parts of the phrase will be highlighted everywhere instead of only when it forms the given phrase.
hl.highlightMultiTerm	true	If set to **true**, Solr will highlight wildcard queries (and other `MultiTermQuery` subclasses). If **false**, they won't be highlighted at all.
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. 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.
hl.payloads	(automatic)	When `hl.usePhraseHighlighter` is true and the indexed field has payloads but not term vectors (generally quite rare), the index's payloads will be read into the highlighter's memory index along with the postings. If this may happen and you know you don't need them for highlighting (i.e. your queries don't filter by payload) then you can save a little memory by setting this to false.

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` param eter 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.useFastVectorHighlighter	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. 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.highlightAlternate	true	If set to **true**, and `hl.alternateFieldName` is active, Solr will show the entire alternate field, with highlighting of occurrences. If `hl.maxAlternateFieldLength=N` is used, Solr returns max `N` characters surrounding the best matching fragment. If set to **false**, or if there is no match in the alternate field either, the alternate field will be shown without highlighting.
hl.tag.pre hl.tag.post	and	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** fr agListBuilder 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 and 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` se ction 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 summaries and efficiency, but is less flexible than the other highlighters. It uses significantly less disk space, 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. 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		Specifies the text that should appear before a highlighted term.

hl.tag.post		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>
    <!-- optional elements with defaults
    <str
name="distanceMeasure">org.apache.lucene.search.spell.LevensteinDistance</str>
    <str name="accuracy">0.5</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 the Schema) 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`. 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, `thresholdTokenFrequency` 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>
    <!-- optional elements with defaults
    <str
name="distanceMeasure">org.apache.lucene.search.spell.LevensteinDistance</str>
    <str name="accuracy">0.5</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. 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>
```

Spell Check Parameters

The SpellCheck component accepts the parameters described in the table below.

Parameter	Description
spellcheck	Turns on or off SpellCheck suggestions for the request. If **true**, then spelling suggestions will be generated.
spellcheck.q 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.collateExtendedResults	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.collateParam.*	Specifies param=value pairs that can be used to override normal query params when validating 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` is 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 `spellcheck.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.collateExtendedResults` Parameter

If **true**, this parameter returns an expanded response format detailing the collations Solr found. The default value is **false** 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.collateParam.*` Parameter Prefix

This parameter prefix can be used to specify any additional parameters that you wish to the Spellchecker to use when internally validating collation queries. For example, even if your regular search results allow for loose matching of one or more query terms via parameters like `"q.op=OR&mm=20%"` you can specify override params such as "`spellcheck.collateParam.q.op=AND&spellcheck.collateParam.mm=100%`" to require that only collations consisting of words that are all found in at least one document may be returned.

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

Using Solr's "`bin/solr -e techproducts`" example, this query shows the results of a simple request that defines a query using the `spellcheck.q` parameter, and forces the collations to require all input terms must match:

```
http://localhost:8983/solr/techproducts/spell?df=text&spellcheck.q=delll+ultra+sharp&spellcheck=true&spellcheck.collateParam.q.op=AND
```

Results:

```
<lst name="spellcheck">
  <lst name="suggestions">
    <lst name="delll">
      <int name="numFound">1</int>
      <int name="startOffset">0</int>
      <int name="endOffset">5</int>
      <int name="origFreq">0</int>
      <arr name="suggestion">
        <lst>
          <str name="word">dell</str>
          <int name="freq">1</int>
        </lst>
      </arr>
    </lst>
    <lst name="ultra sharp">
      <int name="numFound">1</int>
      <int name="startOffset">6</int>
      <int name="endOffset">17</int>
      <int name="origFreq">0</int>
      <arr name="suggestion">
        <lst>
          <str name="word">ultrasharp</str>
          <int name="freq">1</int>
        </lst>
      </arr>
    </lst>
  </lst>
  <bool name="correctlySpelled">false</bool>
  <lst name="collations">
    <lst name="collation">
      <str name="collationQuery">dell ultrasharp</str>
      <int name="hits">1</int>
      <lst name="misspellingsAndCorrections">
        <str name="delll">dell</str>
        <str name="ultra sharp">ultrasharp</str>
      </lst>
    </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/techproducts/spell?spellcheck=true&spellcheck.build=true&spellcheck.q=toyata&shards.qt=/spell&shards=solr-shard1:8983/solr/techpr`

`oducts,solr-shard2:8983/solr/techproducts`

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 wi ll 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.

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
- Available Transformers
  - [value] - ValueAugmenterFactory
  - [explain] - ExplainAugmenterFactory
  - [child] - ChildDocTransformerFactory
  - [shard] - ShardAugmenterFactory
  - [docid] - DocIdAugmenterFactory
  - [elevated] and [excluded]
  - [json] / [xml]
  - [subquery]
    - Subquery Parameters Shift
    - Document field as an input for subquery params
    - Cores and Collections in SolrCloud
  - [geo] - Geospatial formatter

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=t
rue
```

```
"response":{"numFound":32,"start":0,"docs":[
      {
        "id":"6H500F0",
        "[elevated]":true,
        "[excluded]":false},
      {
        "id":"GB18030TEST",
        "[elevated]":false,
        "[excluded]":true},
      {
        "id":"SP2514N",
        "[elevated]":false,
        "[excluded]":false},
...
```

[json] / [xml]

These transformers replace field value containing a string representation of a valid XML or JSON structure with the actual raw XML or JSON structure rather than just the string value. Each applies only to the specific writer, such that `[json]` only applies to `wt=json` and `[xml]` only applies to `wt=xml`.

```
fl=id,source_s:[json]&wt=json
```

[subquery]

This transformer executes a separate query per transforming document passing document fields as an input for subquery parameters. It's usually used with `{!join}` and `{!parent}` query parsers, and is intended to be an improvement for `[child]`.

- It must be given an unique name: `fl=*,children:[subquery]`
- There might be a few of them, eg `fl=*,sons:[subquery],daughters:[subquery]`.
- Every `[subquery]` occurrence adds a field into a result document with the given name, the value of this field is a document list, which is a result of executing subquery using document fields as an input.

Here is how it looks like in various formats:

```
<result name="response" numFound="2" start="0">
      <doc>
         <int name="id">1</int>
         <arr name="title">
            <str>vdczoypirs</str>
         </arr>
         <result name="children" numFound="1" start="0">
            <doc>
               <int name="id">2</int>
               <arr name="title">
                  <str>vdczoypirs</str>
               </arr>
            </doc>
         </result>
      </doc>
  ...
```

```
"response":{
  "numFound":2, "start":0,
  "docs":[
    {
      "id":1,
      "subject":["parentDocument"],
      "title":["xrxvomgu"],
      "children":{
         "numFound":1, "start":0,
         "docs":[
            { "id":2,
              "cat":["childDocument"]
            }
          ]
    }},
    {
       "id":4,
    ...
```

```
SolrDocumentList subResults = (SolrDocumentList)doc.getFieldValue("children");
```

Subquery Parameters Shift

If subquery is declared as `fl=*,foo:[subquery]`, subquery parameters are prefixed with the given name and period. eg

`q=*:*&fl=*,`**`foo`**`:[subquery]&`**`foo.`**`q=to be continued&`**`foo.`**`rows=10&`**`foo.`**`sort=id desc`

Document field as an input for subquery params

It's necessary to pass some document field values as a parameter for subquery. It's supported via implicit **`row.fieldname`** parameter, and can be (but might not only) referred via Local Parameters syntax: `q=namne:john&fl=name,id,depts:[subquery]&depts.q={!terms f=id` **`v=$row.dept_id`**`}&depts.rows=10`

Here departmens are retrieved per every employee in search result. We can say that it's like SQL `join ON emp.dept_id=dept.id`.

Note, when document field has multiple values they are concatenated with comma by default, it can be changed by local parameter `foo:[subquery separator=' ']` , this mimics **`{!terms}`** to work smoothly with it.

Cores and Collections in SolrCloud

Use `foo:[subquery fromIndex=departments]` to invoke subquery on another core on the same node, it's what **`{!join}`** does for non-SolrCloud mode. But in case of SolrCloud just (and only) explicitly specify its' native parameters like `collection, shards` for subquery, eg:

`q=*:*&fl=*,foo:[subquery]&foo.q=cloud&`**`foo.collection`**`=departments`

Note:
Beware that collection should have the same schema.xml, at least a uniqueKey.

[geo] - Geospatial formatter

Formats spatial data from a spatial field using a designated format type name. Two inner parameters are required: `f` for the field name, and `w` for the format name. Example: `geojson:[geo f=mySpatialField w=GeoJSON]`.

Normally you'll simply be consistent in choosing the format type you want by setting the `format` attribute on the spatial field type to `WKT` or `GeoJSON` – see the section Spatial Search for more information. If you are consistent, it'll come out the way you stored it. This transformer offers a convenience to transform the spatial format to something different on retrieval.

In addition, this feature is very useful with the `RptWithGeometrySpatialField` to avoid double-storage of the potentially large vector geometry. This transformer will detect that field type and fetch the geometry from an internal compact binary representation on disk (in docValues), and then format it as desired. As such, you needn't mark the field as stored, which would be redundant. In a sense this double-storage between docValues and stored-value storage isn't unique to spatial but with polygonal geometry it can be a lot of data, and furthermore you'd like to avoid storing it in a verbose format (like GeoJSON or WKT).

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.

Although it is possible to use the Spell Checking functionality to power autosuggest behavior, Solr has a dedicated SuggestComponent designed for this functionality. 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 "`techproducts`" example has the new Suggester implementation configured already. For more on search components, see the section RequestHandlers and SearchComponents in SolrConfig.

Covered in this section:

- Configuring Suggester in solrconfig.xml
  - Adding the Suggest Search Component
  - Adding the Suggest Request Handler
- Example Usages
  - Get Suggestions with Weights
  - Multiple Dictionaries
  - Context Filtering

Configuring Suggester in solrconfig.xml

The "`techproducts`" example `solrconfig.xml` has a `suggest` search component and a `/suggest` request handler 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

The first step is to add a search component to `solrconfig.xml` and tell it to use the SuggestComponent. 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>
    <str name="buildOnStartup">false</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.

Parameter	Description
searchComponent name	Arbitrary name for the search component.
name	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
lookupImpl	Lookup implementation. There are several possible implementations, described below in the section Lookup Implementations. If not set, the default lookup is JaspellLookupFactory.
dictionaryImpl	The dictionary implementation to use. There are several possible implementations, described below in the section Dictionary Implementations. If not set, the default dictionary implementation is HighFrequencyDictionaryFactory unless a `sourceLocation` is used, in which case, the dictionary implementation will be FileDictionaryFactory

field	A field from the index to use as the basis of suggestion terms. If `sourceLocation` is empty (meaning any dictionary implementation other than FileDictionaryFactory) then terms from this field in the index will be used. To be used as the basis for a suggestion, the field must be stored. 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: `<fieldType class="solr.TextField" name="textSuggest" positionIncrementGap="100"> <analyzer> <tokenizer class="solr.StandardTokenizerFactory"/> <filter class="solr.StandardFilterFactory"/> <filter class="solr.LowerCaseFilterFactory"/> </analyzer> </fieldType>` 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.
sourceLocation	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.
storeDir	The location to store the dictionary file.
buildOnCommit or buildOnOptimize	If true then the lookup data structure will be rebuilt after soft-commit. If false, the default, then the lookup data will be built only when requested by URL parameter `suggest.build=true`. Use `buildOnCommit` to rebuild the dictionary with every soft-commit, or `buildOnOptimize` to build the dictionary only when the index is optimized. Some lookup implementations may take a long time to build, specially with large indexes, in such cases, using buildOnCommit or buildOnOptimize, particularly with a high frequency of softCommits is not recommended, and it's recommended instead to build the suggester at a lower frequency by manually issuing requests with `suggest.build=true`.
buildOnStartup	If true then the lookup data structure will be built when Solr starts or when the core is reloaded. If this parameter is not specified, the suggester will check if the lookup data structure is present on disk and build it if not found. Enabling this to true could lead to the core talking longer to load (or reload) as the suggester data structure needs to be built, which can sometimes take a long time. It's usually preferred to have this setting set to 'false' and build suggesters manually issuing requests with `suggest.build=true`.

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.

AnalyzingInfixLookupFactory

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 its 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).
- allTermsRequired: Boolean option for multiple terms. Default is true - all terms required.
- highlight: Highlight suggest terms. Default is true.

This implementation supports Context Filtering.

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. Can be one of:
  - position_linear: weightFieldValue*(1 - 0.10*position): Matches to the start will be given a higher score (Default)
  - position_reciprocal: weightFieldValue/(1+position): Matches to the end will be given a higher score.
    - exponent: an optional configuration variable for the position_reciprocal blenderType used to control how fast the score will increase or decrease. Default 2.0
- 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).

This implementation supports Context Filtering.

FreeTextLookupFactory

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 the following additional properties:

- suggestFreeTextAnalyzerFieldType: The analyzer used at "query-time" and "build-time" to analyze suggestions. This field is required.
- 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 its 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 `spellcheck.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 optional.
- payloadField: The payloadField should be a field that is stored. This field is optional.
- contextField: Field to be used for context filtering. Note that only some lookup implementations support filtering.

DocumentExpressionDictionaryFactory

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:

- 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. This field is required.
- contextField: Field to be used for context filtering. Note that only some lookup implementations support filtering.

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. 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). If using payloads, the first line in the file **must** specify a payload.

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').

Example

```
acquire
accidentally 2.0
accommodate 3.0
```

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">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>
  <lst name="suggester">
    <str name="name">altSuggester</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.cfq	A Context Filter Query used to filter suggestions based on the context field, if supported by the suggester.
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.
suggest.buildAll	If true, it will build all suggester indexes.
suggest.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.



Context Filtering
Context filtering (`suggest.cfq`) is currently only supported by AnalyzingInfixLookupFactory and BlendedInfixLookupFactory, and only when backed by a Document*Dictionary. All other implementations will return unfiltered matches as if filtering was not requested.

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/techproducts/suggest?suggest=true&suggest.build=true&sugg
est.dictionary=mySuggester&wt=json&suggest.q=elec
```

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": {
    "mySuggester": {
      "elec": {
        "numFound": 3,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 2199,
            "payload": ""
          },
          {
            "term": "electronics",
            "weight": 649,
            "payload": ""
          },
          {
            "term": "electronics and stuff2",
            "weight": 279,
            "payload": ""
          }
        ]
      }
    }
  }
}
```

Multiple Dictionaries

If you have defined multiple dictionaries, you can use them in queries.

Example query:

```
http://localhost:8983/solr/techproducts/suggest?suggest=true& \

suggest.dictionary=mySuggester&suggest.dictionary=altSuggester&wt=json&suggest.q=ele
c
```

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": {
    "mySuggester": {
      "elec": {
        "numFound": 1,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 100,
            "payload": ""
          }
        ]
      }
    },
    "altSuggester": {
      "elec": {
        "numFound": 1,
        "suggestions": [
          {
            "term": "electronics and computer1",
            "weight": 10,
            "payload": ""
          }
        ]
      }
    }
  }
}
```

Context Filtering

Context filtering lets you filter suggestions by a separate context field, such as category, department or any other token. The AnalyzingInfixLookupFactory and BlendedInfixLookupFactory currently support this feature, when backed by DocumentDictionaryFactory.

Add `contextField` to your suggester configuration. This example will suggest names and allow to filter by category:

solrconfig.xml

```
<searchComponent name="suggest" class="solr.SuggestComponent">
  <lst name="suggester">
    <str name="name">mySuggester</str>
    <str name="lookupImpl">AnalyzingInfixLookupFactory</str>
    <str name="dictionaryImpl">DocumentDictionaryFactory</str>
    <str name="field">name</str>
    <str name="weightField">price</str>
    <str name="contextField">cat</str>
    <str name="suggestAnalyzerFieldType">string</str>
    <str name="buildOnStartup">false</str>
  </lst>
</searchComponent>
```

Example context filtering suggest query:

```
http://localhost:8983/solr/techproducts/suggest?suggest=true&suggest.build=true& \
   suggest.dictionary=mySuggester&wt=json&suggest.q=c&suggest.cfq=memory
```

The suggester will only bring back suggestions for products tagged with cat=memory.

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.

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.

Covered in this section:

- How MoreLikeThis Works
- Common Parameters for MoreLikeThis
- Parameters for the MoreLikeThisComponent
- Parameters for the MoreLikeThisHandler
- More Like This Query Parser
- Related Topics

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.

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.
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`.

More Like This Query Parser

The `mlt` query parser provides a mechanism to retrieve documents similar to a given document, like the handler. More information on the usage of the mlt query parser can be found here.

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
- 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 – it 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 and 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. If the index is distributed, which is common when running in SolrCloud mode, then 1 million documents are retrieved from **each shard**. For a ten shard index, ten million entries must be retrieved and sorted to figure out the 1000 documents that match those query parameters.

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 criterion.

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 its 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 search for "shoes" on a footwear retailer's e-commerce site, Solr would return all results for that query term, along with selectable facets such as "size," "color," "brand," and so on.

You can however combine grouping with faceting. Grouped faceting supports `facet.field` and `facet.range` but currently doesn't support date and pivot faceting. The facet counts are computed based on the first `group.field` parameter, and other `group.field` parameters are ignored.

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. **NOTE:** This option does not work with distributed searches.
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 `score desc`.
group.sort	sortspec	Specifies how Solr sorts documents within each group. The default behavior if `group.sort` is not specified is to use the same effective value as the `sort` parameter.
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. See below for Distributed Result Grouping Caveats when using sharded indexes
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. See below for Distributed Result Grouping Caveats when using sharded indexes
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.

Examples

All of the following sample queries work with Solr's "`bin/solr -e techproducts`" example.

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/techproducts/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 of the five unique values of `group.field`, Solr returns a `docList` for that `groupValue` such that the `numFound` indicates the total number of documents in that group, and the top documents are returned according to the implicit default `group.limit=1` and `group.sort=score desc` parameters. The resulting groups are then sorted by the score of the top document within each group based on the implicit `sort=score desc`, and the number of groups returned is limited to the implicit `rows=10`.

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 – notably the `numFound` in each group – but it may be easier for existing Solr clients to parse.

```
http://localhost:8983/solr/techproducts/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/techproducts/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 Caveats

Grouping is supported for distributed searches, with some caveats:

- Currently `group.func` is is not supported in any distributed searches
- `group.ngroups` and `group.facet` require that all documents in each group must be co-located on the same shard in order for accurate counts to be returned. Document routing via composite keys can be a useful solution in many situations.

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. The Collapsing query parser groups documents (collapsing the result set) according to your parameters, while the Expand component provides access to documents in the collapsed group for use in results display or other processing by a client application.



In order to use these features with SolrCloud, the documents must be located on the same shard. To ensure document co-location, you can define the `router.name` parameter as `compositeId` when creating the collection. For more information on this option, see the section Document Routing.

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.

The CollapsingQParser accepts the following local parameters:

Parameter	Description	Default
field	The field that is being collapsed on. The field must be a single valued String, Int or Float	none
min | max	Selects the group head document for each group based on which document has the min or max value of the specified numeric field or function query. At most only one of the min, max, or sort (see below) parameters may be specified. If none are specified, the group head document of each group will be selected based on the highest scoring document in that group.	none
sort	Selects the group head document for each group based on which document comes first according to the specified sort string. At most only one of the min, max, (see above) or sort parameters may be specified. If none are specified, the group head document of each group will be selected based on the highest scoring document in that group.	none
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.	ignore
hint	Currently there is only one hint available "`top_fc`", which stands for top level FieldCache. The top_fc hint is only available when collapsing on String fields. top_fc usually provides the best query time speed but takes the longest to warm on startup or following a commit. top_fc will also result in having the collapsed field cached in memory twice if the it's used for faceting or sorting. For very high cardinality (high distinct count) fields, top_fc may not fare so well.	none

size	Sets the initial size of the collapse data structures when collapsing on a **numeric field only**. The data structures used for collapsing grow dynamically when collapsing on numeric fields. Setting the size above the number of results expected in the result set will eliminate the resizing cost.	100,000

Sample Syntax:

Collapse on `group_field` selecting the document in each group with the highest scoring document:

```
fq={!collapse field=group_field}
```

Collapse on `group_field` selecting the document in each group with the minimum value of `numeric_field`:

```
fq={!collapse field=group_field min=numeric_field}
```

Collapse on `group_field` selecting the document in each group with the maximum value of `numeric_field`:

```
fq={!collapse field=group_field max=numeric_field}
```

Collapse on `group_field` selecting the document in each group with the maximum value of a function. Note that the **cscore()** function can be used with the min/max options to use the score of the current document being collapsed.

```
fq={!collapse field=group_field max=sum(cscore(),numeric_field)}
```

Collapse on `group_field` with a null policy so that all docs that do not have a value in the `group_field` will be treated as a single group. For each group, the selected document will be based first on a `numeric_field`, but ties will be broken by score:

```
fq={!collapse field=group_field nullPolicy=collapse sort='numeric_field asc, score
desc'}
```

Collapse on `group_field` with a hint to use the top level field cache:

```
fq={!collapse field=group_field hint=top_fc}
```

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:

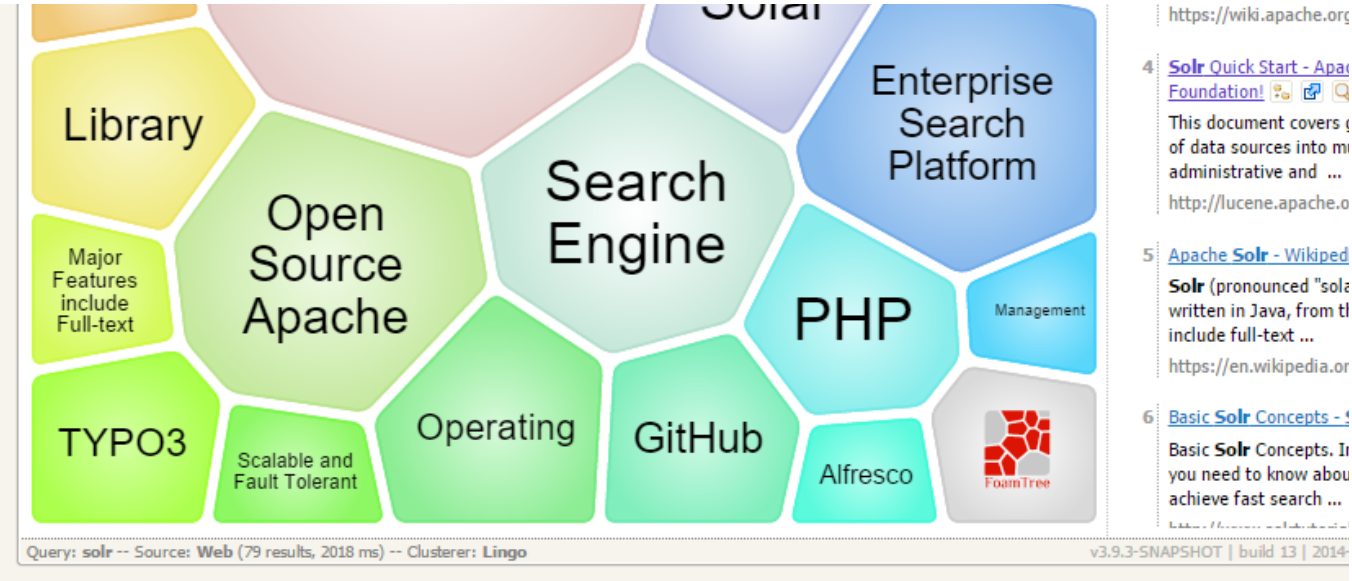
Parameter	Description	Default
expand.sort	Orders the documents within the expanded groups	score desc
expand.rows	The number of rows to display in each group	5
expand.q	Overrides the main q parameter, determines which documents to include in the main group.	main q
expand.fq	Overrides main fq's, determines 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).





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
- Quick Start Example
- Installation
- Configuration
- Tweaking Algorithm Settings
- Performance Considerations
- Additional Resources

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

The "`techproducts`" example included with Solr is pre-configured with all the necessary components for result clustering - but they are disabled by default.

To enable the clustering component contrib and a dedicated search handler configured to use it, specify the "`-a`" option to set a JVM System Property when running the example:

```
bin/solr start -e techproducts -Dsolr.clustering.enabled=true
```

You can now try out the clustering handler by opening the following URL in a browser:

- `http://localhost:8983/solr/techproducts/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 that by default paths are relative to the Solr core so they may need adjustments to your configuration, or an explicit specification of the `$solr.install.dir`.*

```
<lib dir="${solr.install.dir:../../..}/contrib/clustering/lib/"
regex=".*\.jar" />
<lib dir="${solr.install.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 by passing `clustering.engine=(engine name)` URL parameter.*

```
<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.LingoClusterin
gAlgorithm</str>
  </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.STCClusteringAlg
orithm</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`.

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/techproducts/clustering?q=*:*&rows=100&LingoClusteringAlgorithm.desiredClusterCountBase=20.

The clustering engine (the algorithm declared in `solrconfig.xml`) can also be changed at runtime by passing `clustering.engine=name` request attribute: http://localhost:8983/solr/techproducts/clustering?q=*:*&rows=100&clustering.engine=kmeans

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
- Generate a 2D grid of facet count numbers for heatmap generation or point-plotting.

There are three main field types available for spatial search:

- `LatLonType` and its non-geodetic twin `PointType`
- `SpatialRecursivePrefixTreeFieldType` (RPT for short), including `RptWithGeometrySpatialField`, a derivative
- `BBoxField`

RPT offers more features than LatLonType and fast filter performance, although LatLonType is 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. If you need to index shapes other than points (e.g. a circle or polygon) then use RPT.

BBoxField is for indexing bounding boxes, querying by a box, specifying a search predicate (Intersects,Within,Contains,Disjoint,Equals), and a relevancy sort/boost like overlapRatio or simply the area.

Some details that are not in this guide can be found at 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 or GeoJSON.

See the section SpatialRecursivePrefixTreeFieldType below for RPT configuration specifics.

Spatial Filters

There are 2 types of Spatial filters, which both support the following parameters:

Parameter	Description
d	the radial distance, usually in kilometers. (RPT & BBoxField can set other units via the setting `distanceUnits`)
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
score	(Advanced option; RPT and BBoxField field types only) If the query is used in a scoring context (e.g. as the main query in q), this *local parameter* determines what scores will be produced. Valid values are: • `none` - A fixed score of 1.0. (the default) • `kilometers` - distance in kilometers between the field value and the specified center point • `miles` - distance in miles between the field value and the specified center point • `degrees` - distance in degrees between the field value and the specified center point • `distance` - distance between the field value and the specified center point in the `distanceUnits` configured for this field • `recipDistance` - 1 / the distance ⚠ Don't use this for indexed non-point shapes (e.g. polygons). The results will be erroneous. And with RPT, it's only recommended for multi-valued point data, as the implementation doesn't scale very well and for single-valued fields, you should instead use a separate LatLonType field purely for distance sorting. When used with `BBoxField`,additional options are supported: • `overlapRatio` - The relative overlap between the indexed shape & query shape. • `area` - haversine based area of the overlapping shapes expressed in terms of the `distanceUnits` configured for this field • `area2D` - cartesian coordinates based area of the overlapping shapes expressed in terms of the `distanceUnits` configured for this field
filter	(Advanced option; RPT and BBoxField field types only) If you only want the query to score (with the above `score` local parameter), not filter, then set this local parameter to false.

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 parameters. 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
```

RPT

RPT refers to either `SpatialRecursivePrefixTreeFieldType` (aka simply RPT) and an extended version: `RptWithGeometrySpatialField` (aka RPT with Geometry). RPT offers several functional 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 points

- 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), and GeoJSON too. In addition to indexing and searching, this works with the `wt=geojson` (GeoJSON Solr response-writer) and `[geo f=myfield]` (geo Solr document-transformer).
- Heatmap grid faceting capability

RPT incorporates the basic features of LatLonType and PointType, such as lat-lon bounding boxes and circles, in addition to supporting geofilt, bbox, geodist, and a range-queries. RPT with Geometry is defined further below.

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	Solr supports polygons via JTS Topology Suite, which does not come with Solr. 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 `org.locationtech.spatial4j.context.jts.JtsSpatialContextFactory`. *(note: prior to Solr 6, the "org.locationtech.spatial4j" part was "com.spatial4j.core")*. 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 `autoIndex` (i.e. use PreparedGeometry) as it's been shown to be a major performance boost for non-trivial polygons. Further details about specifying polygons to index or query are at Solr's Wiki linked 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.
format	Defines the shape syntax/format to be used. Defaults to `WKT` but `GeoJSON` is another popular format. Spatial4j governs this feature and supports other formats. If a given shape is parseable as "lat,lon" or "x y" then that is always supported.

distanceUnits	This is used to specify the units for distance measurements used throughout the use of this field. This can be `degrees`, `kilometers` or `miles`. It is applied to nearly all distance measurements involving the field: `maxDistErr`, `distErr`, `d`, `geodist` and the `score` when score is `distance`, `area`, or `area2d`. However, it doesn't affect distances embedded in WKT strings, (eg: "`BUFFER(POINT(200 10),0.2)`"), which are still in degrees. `distanceUnits` defaults to either "`kilometers`" if `geo` is "`true`", or "`degress`" if `geo` is "`false`". `distanceUnits` replaces the `units` attribute; which is now deprecated and mutually exclusive with this attribute.
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. At query time this can be overridden in the query syntax, such as to 0.0 so as to not approximate the search shape. The default for the RPT field is 0.025. Note: For RPTWithGeometrySpatialField (see below), there's always complete accuracy with the serialized geometry and so this doesn't control accuracy so much as it controls the trade-off of how big the index should be. distErrPct defaults to 0.15 for that field.
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).
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=true` then the default prefix tree is "`geohash`", otherwise it's "`quad`". Geohash has 32 children at each level, quad has 4. Geohash can only be used for `geo=true` as it's strictly geospatial. A third choice is "`packedQuad`", which is generally more efficient than plain "quad", provided there are many levels -- perhaps 20 or more.
maxLevels	Sets the maximum grid depth for indexed data. Instead, it's usually more intuitive to compute an appropriate maxLevels by specifying `maxDistErr` .

And there are others: `normWrapLongitude` , `datelineRule`, `validationRule`, `autoIndex`, `allowMultiOverlap`, `precisionModel`. For further info, see the note about spatialContextFactory implementations referenced above, especially the link to the JTS based one.

```
<fieldType name="location_rpt"   class="solr.SpatialRecursivePrefixTreeFieldType"

spatialContextFactory="org.locationtech.spatial4j.context.jts.JtsSpatialContextFacto
ry"
               autoIndex="true"
               validationRule="repairBuffer0"
               distErrPct="0.025"
               maxDistErr="0.001"
               distanceUnits="kilometers" />
```

Once the field type has been defined, define a field that uses it.

Here's an example polygon query for a field "geo" that is either solr.SpatialRecursivePrefixTreeFieldType or RptWithGeometrySpatialField:

```
&q=*:*&fq={!field f=geo}Intersects(POLYGON((-10 30, -40 40, -10 -20, 40 20, 0 0,
-10 30)))
```

Inside the parenthesis following the search predicate is the shape definition. The format of that shape is governed by the `format` attribute on the field type, defaulting to WKT. If you prefer GeoJSON, you can specify that instead.

Beyond this reference guide and Spatila4j's docs, there are some details that remain at the Solr Wiki at http://wiki.apache.org/solr/SolrAdaptersForLuceneSpatial4

RptWithGeometrySpatialField

The `RptWithGeometrySpatialField` field type is a derivative of `SpatialRecursivePrefixTreeFieldType` that also stores the original geometry in Lucene DocValues, which it uses to achieve accurate search. It can also be used for indexed point fields. The Intersects predicate (the default) is particularly fast, since many search results can be returned as an accurate hit without requiring a geometry check. This field type is configured just like RPT except that the default `distErrPct` is 0.15 (higher than 0.025) because the grid squares are purely for performance and not to fundamentally represent the shape.

An optional in-memory cache can be defined in `solrconfig.xml`, which should be done when the data tends to have shapes with many vertices. Assuming you name your field "geom", you can configure an optional cache in solrconfig.xml by adding the following – notice the suffix of the cache name:

```
<cache name="perSegSpatialFieldCache_geom"
           class="solr.LRUCache"
           size="256"
           initialSize="0"
           autowarmCount="100%"
           regenerator="solr.NoOpRegenerator"/>
```

When using this field type, you will likely *not* want to mark the field as stored because it's redundant with the DocValues data and surely larger because of the formatting (be it WKT or GeoJSON). To retrieve the spatial data in search results from DocValues, use the `[geo]` transformer -- Transforming Result Documents.

Heatmap Faceting

The RPT field supports generating a 2D grid of facet counts for documents having spatial data in each grid cell. For high-detail grids, this can be used to plot points, and for lesser detail it can be used for heatmap generation. The grid cells are determined at index-time based on RPT's configuration. At facet counting time, the indexed cells in the region of interest are traversed and a grid of counters corresponding to each cell are incremented.

Solr can return the data in a straight-forward 2D array of integers or in a PNG which compresses better for larger data sets but must be decoded.

The heatmap feature is accessed from Solr's faceting feature. As a part of faceting, it supports the `key` local parameter as well as excluding tagged filter queries, just like other types of faceting do. This allows multiple heatmaps to be returned on the same field with different filters.

Parameter	Description
facet	Set to `true` to enable faceting
facet.heatmap	The field name of type RPT
facet.heatmap.geom	The region to compute the heatmap on, specified using the rectangle-range syntax or WKT. It defaults to the world. ex: `["-180 -90" TO "180 90"]`
facet.heatmap.gridLevel	A specific grid level, which determines how big each grid cell is. Defaults to being computed via distErrPct (or distErr)
facet.heatmap.distErrPct	A fraction of the size of geom used to compute gridLevel. Defaults to 0.15. It's computed the same as a similarly named parameter for RPT.
facet.heatmap.distErr	A cell error distance used to pick the grid level indirectly. It's computed the same as a similarly named parameter for RPT.
facet.heatmap.format	The format, either `ints2D` (default) or `png`.

Tip
You'll experiment with different distErrPct values (probably 0.10 - 0.20) with various input geometries till the default size is what you're looking for. The specific details of how it's computed isn't important. For high-detail grids used in point-plotting (loosely one cell per pixel), set distErr to be the number of decimal-degrees of several pixels or so of the map being displayed. Also, you probably don't want to use a geohash based grid because the cell orientation between grid levels flip-flops between being square and rectangle. Quad is consistent and has more levels, albeit at the expense of a larger index.

Here's some sample output in JSON (with some ..... inserted for brevity):

```
{gridLevel=6,columns=64,rows=64,minX=-180.0,maxX=180.0,minY=-90.0,maxY=90.0,
counts_ints2D=[[0, 0, 2, 1, ....],[1, 1, 3, 2, ...],...]}
```

The output shows the gridLevel which is interesting since it's often computed from other parameters. If an interface being developed allows an explicit resolution increase/decrease feature then subsequent requests can specify the gridLevel explicitly.

The `minX`, `maxX`, `minY`, `maxY` reports the region where the counts are. This is the minimally enclosing bounding rectangle of the input `geom` at the target grid level. This may wrap the dateline. The `columns` and `rows` values are how many columns and rows that the output rectangle is to be divided by evenly. Note: Don't divide an on-screen projected map rectangle evenly to plot these rectangles/points since the cell data is in the coordinate space of decimal degrees if geo=true or whatever units were given if geo=false. This could be arranged to be the same as an on-screen map but won't necessarily be.

The `counts_ints2D` key has a 2D array of integers. The initial outer level is in row order (top-down), then the inner arrays are the columns (left-right). If any array would be all zeros, a null is returned instead for efficiency reasons. The entire value is null if there is no matching spatial data.

If `format=png` then the output key is `counts_png`. It's a base-64 encoded string of a 4-byte PNG. The PNG logically holds exactly the same data that the ints2D format does. Note that the alpha channel byte is flipped to

make it easier to view the PNG for diagnostic purposes, since otherwise counts would have to exceed 2^24 before it becomes non-opague. Thus counts greater than this value will become opaque.

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="kilometers" numberType="_bbox_coord"
storeSubFields="false"/>
<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. Alternatively, you could provide a rectangular polygon in WKT (or GeoJSON if you set set `format="GeoJSON"`).

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 parameter can be one of `overlapRatio`, `area`, and `area2D`. `area` scores by the document area using surface-of-a-sphere (assuming `geo=true`) math, while `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. 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. Example: `terms=true`
terms.fl	Yes	null	Specifies the field from which to retrieve terms. Example: `terms.fl=title`

terms.list	No	null	Fetches the document frequency for a comma delimited list of terms. Terms are always returned in index order. Example: terms.list=termA,termB,termC
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. 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. 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. 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). 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). Example: `terms.maxcount=25`
terms.prefix	No	null	Restricts matches to terms that begin with the specified string. 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. Example: `terms.raw=true`
terms.regex	No	null	Restricts matches to terms that match the regular expression. 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: • case_insensitive • comments • multiline • literal • dotall • unicode_case • canon_eq • unix_lines Example: `terms.regex.flag=case_insensitive`
terms.stats	No	null	Include index statistics in the results. Currently returns only the **numDocs** for a collection. When combined with terms.list it provides enough information to compute idf for a list of terms.
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. 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. 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. Example: `terms.upper.incl=true`
terms.list	No	null	Command separated list of terms for which we want the document frequency.

The output is a list of the terms and their document frequency values. See below for examples.

Examples

All of the following sample queries work with Solr's "`bin/solr -e techproducts`" example.

Get Top 10 Terms

This query requests the first ten terms in the name field: `http://localhost:8983/solr/techproducts/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/techproducts/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/techproducts/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/techproducts/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. The examples on this page show how it is configured in Solr's "`techproducts`" example:

```
bin/solr -e techproducts
```

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 the `techproducts` 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 in conjunction with any schema (that has a `uniqueKeyField`) to fetch term vectors for fields configured with the `termVector` attribute, such as in the `techproducts`for example:

```
<field name="includes"
       type="text_general"
       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/techproducts/tvrh?q=*%3A*&start=0&rows=10&fl=id,includes`

```
...
<lst name="termVectors">
  <lst name="GB18030TEST">
    <str name="uniqueKey">GB18030TEST</str>
  </lst>
  <lst name="EN7800GTX/2DHTV/256M">
    <str name="uniqueKey">EN7800GTX/2DHTV/256M</str>
  </lst>
  <lst name="100-435805">
    <str name="uniqueKey">100-435805</str>
  </lst>
  <lst name="3007WFP">
    <str name="uniqueKey">3007WFP</str>
    <lst name="includes">
      <lst name="cable"/>
      <lst name="usb"/>
    </lst>
  </lst>
  <lst name="SOLR1000">
    <str name="uniqueKey">SOLR1000</str>
  </lst>
  <lst name="0579B002">
    <str name="uniqueKey">0579B002</str>
  </lst>
  <lst name="UTF8TEST">
    <str name="uniqueKey">UTF8TEST</str>
  </lst>
  <lst name="9885A004">
    <str name="uniqueKey">9885A004</str>
    <lst name="includes">
      <lst name="32mb"/>
      <lst name="av"/>
      <lst name="battery"/>
      <lst name="cable"/>
      <lst name="card"/>
      <lst name="sd"/>
      <lst name="usb"/>
    </lst>
  </lst>
  <lst name="adata">
    <str name="uniqueKey">adata</str>
  </lst>
  <lst name="apple">
    <str name="uniqueKey">apple</str>
  </lst>
</lst>
```

Request Parameters

The example below shows the available request parameters for this component:

```
http://localhost:8983/solr/techproducts/tvrh?q=includes:[* TO
*]&rows=10&indent=true&tv=true&tv.tf=true&tv.df=true&tv.positions=true&tv.offsets=
true&tv.payloads=true&tv.fl=includes
```

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.payloads	Returns payload information.	boolean
tv.tf	Returns document term frequency info per term in the document.	boolean
tv.tf_idf	Calculates TF / DF (ie: TF * IDF) for each term. Please note that this is a *literal* calculation of "Term Frequency multiplied by Inverse Document Frequency" and **not** a classical TF-IDF similarity measure. 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.

The sample queries in this section assume you are running the "`techproducts`" example included with Solr:

```
bin/solr -e techproducts
```

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. Local Parameters may be used to indicate which subset of the supported statistics should be computed, and/or that statistics should be computed over the results of an arbitrary numeric function (or query) instead of a simple field name. See the examples below.
stats.facet	Returns sub-results for values within the specified facet. This legacy parameter is not recommended for new users - instead please consider combining `stats.field` with `facet.pivot`
stats.calcdistinct	If **true**, the "countDistinct" and "distinctValues" statistics will be computed and included the response. These calculations can be very expensive for fields that do not have a tiny cardinality, so they are disabled by default. This parameter can be specified using per-filed override (ie: `f.<field>.stats.calcdistinct=true`) but users are encouraged to instead the statistics desired as Local Parameter - As a top level request parameter, this option is deprecated.

Example

The query below demonstrates computing stats against two different fields numeric fields, as well as stats over the results of a a 'termfreq()' function call using the 'text' field:

```
http://localhost:8983/solr/techproducts/select?q=*:*&stats=true&stats.field={!func}termfreq('text','memory')&stats.field=price&stats.field=popularity&rows=0&indent=true
```

```
<lst name="stats">
  <lst name="stats_fields">
    <lst name="termfreq(text,memory)">
      <double name="min">0.0</double>
      <double name="max">3.0</double>
      <long name="count">32</long>
      <long name="missing">0</long>
      <double name="sum">10.0</double>
      <double name="sumOfSquares">22.0</double>
      <double name="mean">0.3125</double>
      <double name="stddev">0.7803018439949604</double>
      <lst name="facets"/>
    </lst>
    <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>
    <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>
  </lst>
</lst>
```

Statistics Supported

The table below explains the statistics supported by the Stats component. Not all statistics are supported for all field types, and not all statistics are computed by default (See Local Parameters below for details)

Local Param	Sample Input	Description	Supported Types	Computed by Default
min	true	The minimum value of the field/function in all documents in the set.	All	Yes
max	true	The maximum value of the field/function in all documents in the set.	All	Yes
sum	true	The sum of all values of the field/function in all documents in the set.	Numeric & Date	Yes

count	true	The number of values found in all documents in the set for this field/function.	All	Yes
missing	true	The number of documents in the set which do not have a value for this field/function.	All	Yes
sumOfSquares	true	Sum of all values squared (a by product of computing stddev)	Numeric & Date	Yes
mean	true	The average `(v1 + v2 .... + vN)/N`	Numeric & Date	Yes
stddev	true	Standard deviation, measuring how widely spread the values in the data set are.	Numeric & Date	Yes
percentiles	"1,99,99.9"	A list of percentile values based on cut-off points specified by the param value. These values are an approximation, using the t-digest algorithm.	Numeric	No
distinctValues	true	The set of all distinct values for the field/function in all of the documents in the set. This calculation can be very expensive for fields that do not have a tiny cardinality.	All	No
countDistinct	true	The exact number of distinct values in the field/function in all of the documents in the set. This calculation can be very expensive for fields that do not have a tiny cardinality.	All	No
cardinality	"true" or "0.3"	A statistical approximation (currently using the Hyper LogLog algorithm) of the number of distinct values in the field/function in all of the documents in the set. This calculation is much more efficient then using the 'countDistinct' option, but may not be 100% accurate. Input for this option can be floating point number between 0.0 and 1.0 indicating how aggressively the algorithm should try to be accurate: 0.0 means use as little memory as possible; 1.0 means use as much memory as needed to be as accurate as possible. 'true' is supported as an alias for "0.3"	All	No

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`
- Tagging stats for use with `facet.pivot`: `stats.field={!tag=my_pivot_stats}price`

Local parameters can also be used to specify individual statistics by name, overriding the set of statistics computed by default, eg: `stats.field={!min=true max=true percentiles='99,99.9,99.99'}price`



If any supported statistics are specified via local parameters, then the entire set of default statistics is overridden and only the requested statistics are computed.

Additional "Expert" local params are supported in some cases for affecting the behavior of some statistics:

- `percentiles`
  - `tdigestCompression` - a positive numeric value defaulting to `100.0` controlling the compression factor of the T-Digest. Larger values means more accuracy, but also uses more memory.
- `cardinality`
  - `hllPreHashed` - a boolean option indicating that the statistics are being computed over a "long" field that has already been hashed at index time – allowing the HLL computation to skip this step.
  - `hllLog2m` - an integer value specifying an explicit "log2m" value to use, overriding the heuristic value determined by the cardinality local param and the field type – see the java-hll documentation for more details
  - `hllRegwidth` - an integer value specifying an explicit "regwidth" value to use, overriding the heuristic value determined by the cardinality local param and the field type – see the java-hll docum entation for more details
- `calcDistinct` - for backwards compatibility, `calcDistinct=true` may be specified as an alias for both `countDistinct=true distinctValues=true`

Examples

Here we compute some statistics for the price field. The min, max, mean, 90th, and 99th percentile price values are computed against all products that are in stock (`q=*:*` and `fq=inStock:true`), and independently all of the default statistics are computed against all products regardless of whether they are in stock or not (by excluding that filter).

`http://localhost:8983/solr/techproducts/select?q=*:*&fq={!tag=stock_check}inStock:true&stats=true&stats.field={!ex=stock_check+key=instock_prices+min=true+max=true+mean=true+percentiles='90,99'}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>
      <double name="mean">328.20437693595886</double>
      <lst name="percentiles">
        <double name="90.0">564.9700012207031</double>
        <double name="99.0">1966.6484985351556</double>
      </lst>
    </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>
  </lst>
</lst>
```

The Stats Component and Faceting

Although the `stats.facet` parameter is no longer recommended, sets of `stats.field` parameters can be referenced by '`tag`' when using Pivot Faceting to compute multiple statistics at every level (i.e.: field) in the tree of pivot constraints.

For more information and a detailed example, please see Combining Stats Component With Pivots.

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.

All of the sample configuration and queries used in this section assume you are running Solr's "`techproducts`" example:

```
bin/solr -e techproducts
```

Configuring the Query Elevation Component

You can configure the Query Elevation Component in the `solrconfig.xml` file.

```
<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>`. 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="foo bar">
    <doc id="1" />
    <doc id="2" />
    <doc id="3" />
  </query>

  <query text="ipod">
    <doc id="MA147LL/A" />  <!-- put the actual ipod at the top -->
    <doc id="IW-02" exclude="true" /> <!-- exclude this cable -->
  </query>
</elevate>
```

In this example, the query "foo bar" would first return documents 1, 2 and 3, then whatever normally appears for the same query. For the query "ipod", it would first return "MA147LL/A", and would make sure that "IW-02" 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/techproducts/elevate?q=ipod&df=text&debugQuery=true&enableElevation=true`

`http://localhost:8983/solr/techproducts/elevate?q=ipod&df=text&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/techproducts/elevate?q=ipod&df=text&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/techproducts/elevate?q=ipod&df=text&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/techproducts/elevate?q=ipod&df=text&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/techproducts/elevate?q=ipod&df=text&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 3007WFP and 9885A004 will be elevated, and document IW-02 will be excluded -- regardless of what elevations or exclusions are configured for the query "cable" in elevate.xml:

```
http://localhost:8983/solr/techproducts/elevate?q=cable&df=text&excludeIds=IW-02&elevateIds=3007WFP,9885A004
```

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 IW-02 and F8V7067-APL-KIT will be elevated, and no documents will be excluded – regardless of what elevations or exclusions are configured for the query "ipod" in elevate.xml:

```
http://localhost:8983/solr/techproducts/elevate?q=ipod&df=text&elevateIds=IW-02,F8V7067-APL-KIT
```

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
geojson	GeoJSONResponseWriter
json	JSONResponseWriter
php	PHPResponseWriter
phps	PHPSerializedResponseWriter
python	PythonResponseWriter
ruby	RubyResponseWriter
smile	SmileResponseWriter
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 if the Solr server is upgraded and a new default format is introduced.

Currently supported version values are:

XML Version	Notes
2.2	The format of the responseHeader changed to use the same `<lst>` structure as the rest of the response.

The default value is the latest supported.

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 `sample_techproducts_configs` config set in the Solr distribution, 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.

The default mime type for the JSON writer is `application/json`, however this can be overridden in the `solr config.xml` - such as in this example from the "`techproducts`" configuration:

```
<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>
```

GeoJSON Response Writer

Returns Solr results in GeoJSON augmented with Solr-specific JSON. To use this, set `wt=geojson` and `geojson.field` to the name of a spatial Solr field. Not all spatial fields types are supported, and you'll get an error if you use an unsupported one.

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:

```
$code =
file_get_contents('http://localhost:8983/solr/techproducts/select?q=iPod&wt=php');
eval("$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/techproducts/select?q=iPod&wt=phps');
$result = unserialize($serializedResult);
print_r($result);
```

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/techproducts/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, as well as psuedo-fields, and the output of this CSV format is compatible with Solr's CSV update format.

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/techproducts/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` processes the Solr response and request context through Apache Velocity templating.

See Velocity Response Writer section for details.

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.

Smile Response Writer

The Smile format is a JSON-compatible binary format, described in detail here: http://wiki.fasterxml.com/SmileFormat.

Velocity Response Writer

The VelocityResponseWriter is an optional plugin available in the `contrib/velocity` directory. It powers the /browse user interfaces when using configurations such as "basic_configs", "techproducts", and "example/files".

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">
  <str name="template.base.dir">${velocity.template.base.dir:}</str>

<!--
  <str name="init.properties.file">velocity-init.properties</str>
  <bool name="params.resource.loader.enabled">true</bool>
  <bool name="solr.resource.loader.enabled">false</bool>
  <lst name="tools">
    <str name="mytool">com.example.MyCustomTool</str>
  </lst>
-->
</queryResponseWriter>
```

The above example shows the optional initialization and custom tool parameters used by VelocityResponseWriter; these are detailed in the following table. These initialization parameters are only specified in the writer registration in solrconfig.xml, not as request-time parameters. See further below for request-time parameters.

VelocityResponseWriter initialization parameters

Parameter	
template.base.dir	If specified and exists as a file system directory, a file resource loader will be a templates.
init.properties.file	Specifies a properties file name which must exist in the Solr `conf/` directory (r
params.resource.loader.enabled	The "params" resource loader allows templates to be specified in Solr request `http://localhost:8983/solr/gettingstarted/select?q=*:*&wt=` where `v.template=custom` says to render a template called "custom" and `v` a niche, unusual, use case to need this enabled.
solr.resource.loader.enabled	The "solr" resource loader is the only template loader registered by default. Ter bdirectory. The VelocityResponseWriter itself has some built-in templates (in its templates can be overridden when the same template name is in conf/velocity/
tools	External "tools" can be specified as list of string name/value (tool name / class constructed using a no-arg constructor (or a single-SolrCore-arg constructor if i can override the built-in context objects with the same name, except for $reque

VelocityResponseWriter request parameters

Parameter	Description	Default value
v.template	Specifies the name of the template to render.	
v.layout	Specifies a template name to use as the layout around the main, `v.template`, specified template. The main template is rendered into a string value included into the layout rendering as `$c ontent`.	

v.layout.enabled	Determines if the main template should have a layout wrapped around it. True by default, but requires `v.layout` to specified as well.	true
v.contentType	Specifies the content type used in the HTTP response. If not specified, the default will depend on whether `v.json` is specified or not.	without json.wrf: text/html;charset=UTF-8 with json.wrf: application/json;charset=UTF-8
v.json	Specifies a function name to wrap around the response rendered as JSON. If specified, the content type used in the response will be "application/json;charset=UTF-8", unless overridden by `v.contentType`. Output will be in this format (with v.json=wrf): `wrf("result":"<Velocity generated response string, with quotes and backslashes escaped>")`	
v.locale	Locale to use with the `$resource` tool and other LocaleConfig implementing tools. The default locale is `Locale.ROOT`. Localized resources are loaded from standard Java resource bundles named `resources[_locale-code].properties`. Resource bundles can be added by providing a JAR file visible by the SolrResourceLoader with resource bundles under a velocity sub-directory. Resource bundles are not loadable under conf/, as only the class loader aspect of SolrResourceLoader can be used here.	
v.template.<template_name>	When the "params" resource loader is enabled, templates can be specified as part of the Solr request.	

VelocityResponseWriter context objects

Context reference	Description
request	SolrQueryRequest javadocs
response	QueryResponse most of the time, but in some cases where QueryResponse doesn't like the request handlers output (AnalysisRequestHandler, for example, causes a ClassCastException parsing "response"), the response will be a SolrResponseBase object.
esc	A Velocity EscapeTool instance
date	A Velocity ComparisonDateTool instance
list	A Velocity ListTool instance

math	A Velocity MathTool instance
number	A Velocity NumberTool instance
sort	A Velocity SortTool instance
display	A Velocity DisplayTool instance
resource	A Velocity ResourceTool instance
engine	The current VelocityEngine instance
page	An instance of Solr's PageTool (only included if the response is a QueryResponse where paging makes sense)
debug	A shortcut to the debug part of the response, or null if debug is not on. This is handy for having debug-only sections in a template using `#if($debug)...#end`
content	The rendered output of the main template, when rendering the layout (v.layout.enabled=true and v.layout=<template>).
[custom tool(s)]	Tools provided by the optional "tools" list of the VelocityResponseWriter registration are available by their specified name.

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 does not block 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`	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/my_collection/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/my_collection/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 (`waitFlush=false`):

```
curl
'http://localhost:8983/solr/my_collection/update?optimize=true&maxSegments=10&waitFl
ush=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/my_collection/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.

Real Time Get relies on the update log feature, which is enabled by default and can be configured in `solrconfig.xml`:

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
</updateLog>
```

Real Time Get requests can be performed using the `/get` handler which exists implicitly in Solr - it's equivalent to the following configuration:

```
<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>
```

For example, if you started Solr using the `bin/solr -e techproducts` example command, you could then index a new document (with out committing it) like so:

```
curl 'http://localhost:8983/solr/techproducts/update/json?commitWithin=10000000'
  -H 'Content-type:application/json' -d '[{"id":"mydoc","name":"realtime-get
test!"}]'
```

If you do a normal search, this document should not be found yet:

```
http://localhost:8983/solr/techproducts/query?q=id:mydoc
...
"response":
{"numFound":0,"start":0,"docs":[]}
```

However if you use the Real Time Get handler exposed at `/get`, you can still retrieve that document:

```
http://localhost:8983/solr/techproducts/get?id=mydoc
...
{"doc":{"id":"mydoc","name":"realtime-get test!", "_version_":1487137811571146752}}
```

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.

For example:

```
http://localhost:8983/solr/techproducts/get?ids=mydoc,IW-02
http://localhost:8983/solr/techproducts/get?id=mydoc&id=IW-02
...
{"response":
  {"numFound":2,"start":0,"docs":
    [ { "id":"mydoc",
        "name":"realtime-get test!",
        "_version_":1487137811571146752},
      {
        "id":"IW-02",
        "name":"iPod & iPod Mini USB 2.0 Cable",
        ...
    ]
 }
}
```

Real Time Get requests can also be combined with filter queries, specified with an `fq` parameter, just like search requests:

```
http://localhost:8983/solr/techproducts/get?id=mydoc&id=IW-02&fq=name:realtime-get
...
{"response":
  {"numFound":1,"start":0,"docs":
    [ { "id":"mydoc",
        "name":"realtime-get test!",
        "_version_":1487137811571146752}
    ]
 }
}
```



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

It's possible to export fully sorted result sets using a special rank query parser and response writer specifically designed to work together to handle scenarios that involve sorting and exporting millions of records. This uses a stream sorting technique that begins to send records 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.

Defining the /export Request Handler

To export the full sorted result set you'll want to use a request handler explicitly configured to only run the "query" component, using the the export "rq" and "wt" params.

An "`/export`" request handler with the appropriate configuration is included in the `techproducts` example so `lrconfig.xml`. If however, you would like to add it to an 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.

Here is an example of what an export request of some indexed log data might look like:

```
http://localhost:8983/solr/core_name/export?q=my-query&sort=severity+desc,timestamp+
desc&fl=severity,timestamp,msg
```

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, date, boolean) 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

See the section Streaming Expressions for distributed support.

Streaming Expressions

Streaming Expressions provide a simple yet powerful stream processing language for SolrCloud. They are a suite of functions that can be combined to perform many different parallel computing tasks. These functions are the basis for the Parallel SQL Interface.

There are several available functions, including those that implement:

- continuous push streaming
- continuous pull streaming
- request/response streaming
- MapReduce shuffling aggregation
- pushdown faceted aggregation
- parallel relational algebra (distributed joins, intersections, unions, complements)
- publish/subscribe messaging
- distributed graph traversal
- parallel iterative machine learning

Streams from outside systems can be joined with streams originating from Solr and users can add their own stream functions by following Solr's Java streaming API.



Both streaming expressions and the streaming API are considered experimental, and the APIs are subject to change.

- Stream Language Basics
  - Streaming Requests and Responses
  - Data Requirements
- Stream Sources
  - search
  - jdbc
  - facet
  - features
  - gatherNodes
  - random
  - shortestPath
  - stats
  - train
  - topic
- Stream Decorators
  - complement
  - daemon
  - leftOuterJoin
  - hashJoin
  - innerJoin
  - intersect
  - merge
  - outerHashJoin
  - parallel
  - reduce
  - rollup
  - scoreNodes
  - select
  - sort
  - top
  - unique
  - update

Stream Language Basics

Streaming Expressions are comprised of streaming functions which work with a Solr collection. They emit a stream of tuples (key/value Maps).

Many of the provided streaming functions are designed to work with entire result sets rather then the top N results like normal search. This is supported by the /export handler.

Some streaming functions act as stream sources to originate the stream flow. Other streaming functions act as stream decorators to wrap other stream functions and perform operations on the stream of tuples. Many streams functions can be parallelized across a worker collection. This can be particularly powerful for relational algebra functions.

Streaming Requests and Responses

Solr has a `/stream` request handler that takes streaming expression requests and returns the tuples as a JSON stream. This request handler is implicitly defined, meaning there is nothing that has to be defined in `solrconfig.xml`.

The `/stream` request handler takes one parameter, `expr`, which is used to specify the streaming expression. For example, this curl command encodes and POSTs a simple `search()` expression to the `/stream` handler:

```
curl --data-urlencode 'expr=search(enron_emails,
                                   q="from:1800flowers*",
                                   fl="from, to",
                                   sort="from asc",
                                   qt="/export")'
http://localhost:8983/solr/enron_emails/stream
```

Details of the parameters for each function are included below.

For the above example the `/stream` handler responded with the following JSON response:

```
{"result-set":{"docs":[
   {"from":"1800flowers.133139412@s2u2.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers.93690065@s2u2.com","to":"jtholt@ect.enron.com"},
   {"from":"1800flowers.96749439@s2u2.com","to":"alewis@enron.com"},
   {"from":"1800flowers@1800flowers.flonetwork.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@1800flowers.flonetwork.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@1800flowers.flonetwork.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@1800flowers.flonetwork.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@1800flowers.flonetwork.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"ebass@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"lcampbel@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"ebass@enron.com"},
   {"from":"1800flowers@shop2u.com","to":"ebass@enron.com"},
   {"EOF":true,"RESPONSE_TIME":33}]}
}
```

Note the last tuple in the above example stream is `{"EOF":true,"RESPONSE_TIME":33}`. The `EOF` indicates the end of the stream. To process the JSON response, you'll need to use a streaming JSON implementation because streaming expressions are designed to return the entire result set which may have millions of records.

In your JSON client you'll need to iterate each doc (tuple) and check for the EOF tuple to determine the end of stream.

The `org.apache.solr.client.solrj.io` package provides Java classes that compile streaming expressions into streaming API objects. These classes can be used to execute streaming expressions from inside a Java application. For example:

```
StreamFactory streamFactory = new
StreamFactory().withCollectionZkHost("collection1", zkServer.getZkAddress())
    .withStreamFunction("search", CloudSolrStream.class)
    .withStreamFunction("unique", UniqueStream.class)
    .withStreamFunction("top", RankStream.class)
    .withStreamFunction("group", ReducerStream.class)
    .withStreamFunction("parallel", ParallelStream.class);

ParallelStream pstream =
(ParallelStream)streamFactory.constructStream("parallel(collection1,
group(search(collection1, q=\"*:*\", fl=\"id,a_s,a_i,a_f\", sort=\"a_s asc,a_f
asc\", partitionKeys=\"a_s\"), by=\"a_s asc\"), workers=\"2\",
zkHost=\""+zkHost+"\", sort=\"a_s asc\")");
```

Data Requirements

Because streaming expressions relies on the `/export` handler, many of the field and field type requirements to use `/export` are also requirements for `/stream`, particularly for `sort` and `fl` parameters. Please see the section Exporting Result Sets for details.

Stream Sources

Stream sources originate streams. There are several stream sources available: **search**, **jdbc**, **facet**, **gatherNodes**, **random**, **stats**, **topic**, and **shortestPath**.

search

The `search` function searches a SolrCloud collection and emits a stream of tuples that match the query. This is very similar to a standard Solr query, and uses many of the same parameters.

This expression allows you to specify a request hander using the `qt` parameter. By default, the `/select` handler is used. The `/select` handler can be used for simple rapid prototyping of expressions. For production, however, you will most likely want to use the `/export` handler which is designed to `sort` and `export` entire result sets. The `/export` handler is not used by default because it has stricter requirements then the `/select` handler so it's not as easy to get started working with. To read more about the `/export` handler requirements review the section Exporting Result Sets.

Parameters

- `collection`: (Mandatory) the collection being searched.
- `q`: (Mandatory) The query to perform on the Solr index.
- `fl`: (Mandatory) The list of fields to return.
- `sort`: (Mandatory) The sort criteria.
- `zkHost`: Only needs to be defined if the collection being searched is found in a different zkHost than the local stream handler.
- `qt`: Specifies the query type, or request handler, to use. Set this to `/export` to work with large result sets. The default is `/select`.

- `rows`: (Mandatory with the `/select` handler) The rows parameter specifies how many rows to return. This parameter is only needed with the `/select` handler (which is the default) since the `/export` handle r always returns all rows.
- partitionKeys: Comma delimited list of keys to partition the search results by. To be used with the parallel function for parallelizing operations across worker nodes. See the parallel function for details.

Syntax

```
expr=search(collection1,
       zkHost="localhost:9983",
       qt="/export",
       q="*:*",
       fl="id,a_s,a_i,a_f",
       sort="a_f asc, a_i asc")
```

jdbc

The `jdbc` function searches a JDBC datasource and emits a stream of tuples representing the JDBC result set. Each row in the result set is translated into a tuple and each tuple contains all the cell values for that row.

Parameters

- `connection`: (Mandatory) JDBC formatted connection string to whatever driver you are using.
- `sql`: (Mandatory) query to pass off to the JDBC endpoint
- `sort`: (Mandatory) The sort criteria indicating how the data coming out of the JDBC stream is sorted
- `driver`: The name of the JDBC driver used for the connection. If provided then the driver class will attempt to be loaded into the JVM. If not provided then it is assumed that the driver is already loaded into the JVM. Some drivers require explicit loading so this option is provided.
- `[driverProperty]`: One or more properties to pass to the JDBC driver during connection. The format is `propertyName="propertyValue"`. You can provide as many of these properties as you'd like and they will all be passed to the connection.

Connections and Drivers

Because some JDBC drivers require explicit loading the `driver` parameter can be used to provide the driver class name. If provided, then during stream construction the driver will be loaded. If the driver cannot be loaded because the class is not found on the classpath, then stream construction will fail.

When the JDBC stream is opened it will validate that a driver can be found for the provided connection string. If a driver cannot be found (because it hasn't been loaded) then the open will fail.

Datatypes

Due to the inherent differences in datatypes across JDBC sources the following datatypes are supported. The table indicates what Java type will be used for a given JDBC type. Types marked as requiring conversion will go through a conversion for each value of that type. For performance reasons the cell data types are only considered when the stream is opened as this is when the converters are created.

JDBC Type	Java Type	Requires Conversion
String	String	No
Short	Long	Yes
Integer	Long	Yes

Long	Long	No
Float	Double	Yes
Double	Double	No
Boolean	Boolean	No

Syntax

A basic `jdbc` expression:

```
jdbc(
 connection="jdbc:hsqldb:mem:.",
 sql="select NAME, ADDRESS, EMAIL, AGE from PEOPLE where AGE > 25 order by AGE, NAME
DESC",
 sort="AGE asc, NAME desc",
 driver="org.hsqldb.jdbcDriver"
)
```

A `jdbc` expression that passes a property to the driver:

```
// get_column_name is a property to pass to the hsqldb driver
jdbc(
 connection="jdbc:hsqldb:mem:.",
 sql="select NAME as FIRST_NAME, ADDRESS, EMAIL, AGE from PEOPLE where AGE > 25
order by AGE, NAME DESC",
 sort="AGE asc, NAME desc",
 driver="org.hsqldb.jdbcDriver",
 get_column_name="false"
)
```

facet

The `facet` function provides aggregations that are rolled up over buckets. Under the covers the facet function pushes down the aggregation into the search engine using Solr's JSON Facet API. This provides sub-second performance for many use cases. The facet function is appropriate for use with a low to moderate number of distinct values in the bucket fields. To support high cardinality aggregations see the rollup function.

Parameters

- `collection`: (Mandatory) Collection the facets will be aggregated from.
- `q`: (Mandatory) The query to build the aggregations from.
- `buckets`: (Mandatory) Comma separated list of fields to rollup over. The comma separated list represents the dimensions in a multi-dimensional rollup.
- `bucketSorts`: Comma separated list of sorts to apply to each dimension in the buckets parameters. Sorts can be on the computed metrics or on the bucket values.
- `bucketSizeLimit`: The number of buckets to include. This value is applied to each dimension.
- `metrics`: List of metrics to compute for the buckets. Currently supported metrics are `sum(col)`, `avg(col)`, `min(col)`, `max(col)`, `count(*)`.

Syntax

Example 1:

```
facet(collection1,
      q="*:*",
      buckets="a_s",
      bucketSorts="sum(a_i) desc",
      bucketSizeLimit=100,
      sum(a_i),
      sum(a_f),
      min(a_i),
      min(a_f),
      max(a_i),
      max(a_f),
      avg(a_i),
      avg(a_f),
      count(*))
```

The example above shows a facet function with rollups over a single bucket, where the buckets are returned in descending order by the calculated value of the `sum(a_i)` metric.

Example 2:

```
facet(collection1,
      q="*:*",
      buckets="year_i, month_i, day_i",
      bucketSorts="year_i desc, month_i desc, day_i desc",
      bucketSizeLimit=100,
      sum(a_i),
      sum(a_f),
      min(a_i),
      min(a_f),
      max(a_i),
      max(a_f),
      avg(a_i),
      avg(a_f),
      count(*))
```

The example above shows a facet function with rollups over three buckets, where the buckets are returned in descending order by bucket value.

features

The `features` function extracts the key terms from a text field in a classification training set stored in a SolrCloud collection. It uses an algorithm known as **Information Gain**, to select the important terms from the training set. The `features` function was designed to work specifically with the train function, which uses the extracted features to train a text classifier.

The `features` function is designed to work with a training set that provides both positive and negative examples of a class. It emits a Tuple for each feature term that is extracted along with the inverse document frequency (IDF) for the term in the train set.

The `features` function uses a query to select the training set from a collection. The IDF for each selected feature is calculated relative to the training set matching the query. This allows multiple training sets to be stored in the same SolrCloud collection without polluting the IDF across training sets.

Parameters

- `collection`: (Mandatory) The collection that holds the training set

- `q`: (Mandatory) The query that defines the training set. The idf for the features will be generated on the
- `featureSet`: (Mandatory) The name of the feature set. This can be used to retrieve the features if they are stored in a Solr Cloud collection.
- `field`: (Mandatory) The text field to extract the features from.
- `outcome`: (Mandatory) The field that defines the class, positive or negative
- `numTerms`: (Mandatory) How many feature terms to extract.
- `positiveLabel`: (defaults to 1) The value in the outcome field that defines a postive outcome.

Syntax

```
features(collection1,
         q="*:*",
         featureSet="features1",
         field="body",
         outcome="out_i",
         numTerms=250)
```

gatherNodes

The `gatherNodes` function provides breadth-first graph traversal. For details, see the section Graph Traversal.

random

The `random` function searches a SolrCloud collection and emits a pseudo-random set of results that match the query. Each invocation of random will return a different pseudo-random result set.

Parameters

- `collection`: (Mandatory) The collection the stats will be aggregated from.
- `q`: (Mandatory) The query to build the aggregations from.
- `rows`: (Mandatory) The number of pseudo-random results to return.
- fl: (Mandatory) The field list to return.
- `fq`: (Optional) Filter query

Syntax

```
random(baskets,
       q="productID:productX",
       rows="100",
       fl="basketID")
```

In the example above the `random` function is searching the baskets collections for all rows where "productID:productX". It will return 100 pseudo-random results. The field list returned is the basketID.

shortestPath

The `shortestPath` function is an implementation of a shortest path graph traversal. The `shortestPath` functi on performs an iterative breadth-first search through an unweighted graph to find the shortest paths between two nodes in a graph. The `shortestPath` function emits a tuple for each path found. Each tuple emitted will contain a `path` key which points to a `List` of nodeIDs comprising the path.

Parameters

- `collection`: (Mandatory) The collection that the topic query will be run on.
- `from`: (Mandatory) The nodeID to start the search from
- `to`: (Mandatory) The nodeID to end the search at
- `edge`: (Mandatory) Syntax: `from_field=to_field`. The `from_field` defines which field to search from. The `to_field` defines which field to search to. See example below for a detailed explanation.
- `threads`: (Optional : Default 6) The number of threads used to perform the partitioned join in the traversal.
- `partitionSize`: (Optional : Default 250) The number of nodes in each partition of the join.
- `fq`: (Optional) Filter query
- `maxDepth`: (Mandatory) Limits to the search to a maximum depth in the graph.

Syntax

```
shortestPath(collection,
             from="john@company.com",
             to="jane@company.com",
             edge="from_address=to_address",
             threads="6",
             partitionSize="300",
             fq="limiting query",
             maxDepth="4")
```

The expression above performs a breadth-first search to find the shortest paths in an unweighted, directed graph.

The search starts from the nodeID "john@company.com" in the `from_address` field and searches for the nodeID "jane@company.com" in the `to_address` field. This search is performed iteratively until the `maxDepth` has been reached. Each level in the traversal is implemented as a parallel partitioned nested loop join across the entire collection. The `threads` parameter controls the number of threads performing the join at each level, while the `partitionSize` parameter controls the of number of nodes in each join partition. The `maxDepth` parameter controls the number of levels to traverse. `fq` is a limiting query applied to each level in the traversal.

stats

The `stats` function gathers simple aggregations for a search result set. The stats function does not support rollups over buckets, so the stats stream always returns a single tuple with the rolled up stats. Under the covers the stats function pushes down the generation of the stats into the search engine using the StatsComponent. The stats function currently supports the following metrics: `count(*)`, `sum()`, `avg()`, `min()`, and `max()`.

Parameters

- `collection`: (Mandatory) Collection the stats will be aggregated from.
- `q`: (Mandatory) The query to build the aggregations from.
- `metrics`: (Mandatory) The metrics to include in the result tuple. Current supported metrics are `sum(col)`, `avg(col)`, `min(col)`, `max(col)` and `count(*)`

Syntax

```
stats(collection1,
      q=*:*,
      sum(a_i),
      sum(a_f),
      min(a_i),
      min(a_f),
      max(a_i),
      max(a_f),
      avg(a_i),
      avg(a_f),
      count(*))
```

train

The `train` function trains a Logistic Regression text classifier on a training set stored in a Solr Cloud collection. It uses a parallel iterative, batch Gradient Descent approach to train the model. The training algorithm is embedded inside Solr so with each iteration only the model is streamed across the network.

The `train` function wraps a features function which provides the terms and inverse document frequency (IDF) used to train the model. The `train` function operates over the same training set as the `features` function, which includes both positive and negative examples of the class.

With each iteration the `train` function emits a Tuple with the model. The model contains the feature terms, weights, and the confusion matrix for the model. The optimized model can then be used to classify documents based on their feature terms.

Parameters

- `collection`: (Mandatory) Collection that holds the training set
- `q`: (Mandatory) The query that defines the training set. The idf for the features will be generated on the
- `name`: (Mandatory) The name of model. This can be used to retrieve the model if they stored in a Solr Cloud collection.
- `field`: (Mandatory) The text field to extract the features from.
- `outcome`: (Mandatory) The field that defines the class, positive or negative
- `maxIterations`: (Mandatory) How many training iterations to perform.
- `positiveLabel`: (defaults to 1) The value in the outcome field that defines a positive outcome.

Syntax

```
train(collection1,
      features(collection1, q="*:*", featureSet="first", field="body",
outcome="out_i", numTerms=250),
      q="*:*",
      name="model1",
      field="body",
      outcome="out_i",
      maxIterations=100)
```

topic

The `topic` function provides publish/subscribe messaging capabilities built on top of SolrCloud. The topic function allows users to subscribe to a query. The function then provides one-time delivery of new or updated documents that match the topic query. The initial call to the topic function establishes the checkpoints for the

specific topic ID. Subsequent calls to the same topic ID will return documents added or updated after the initial checkpoint. Each run of the topic query updates the checkpoints for the topic ID. Setting the initialCheckpoint parameter to 0 will cause the topic to process all documents in the index that match the topic query.



The topic function should be considered in beta until SOLR-8709 is committed and released.

Parameters

- `checkpointCollection`: (Mandatory) The collection where the topic checkpoints are stored.
- `collection`: (Mandatory) The collection that the topic query will be run on.
- `id`: (Mandatory) The unique ID for the topic. The checkpoints will be saved under this id.
- `q`: (Mandatory) The topic query.
- `fl`: (Mandatory) The field list returned by the topic function.
- initialCheckpoint: (Optional) Sets the initial Solr _version_ number to start reading from the queue. If not set defaults to the highest version in the index. Setting to 0 will process all records that match query in the index.

Syntax

```
topic(checkpointCollection,
      collection,
      id="uniqueId",
      q="topic query",
      fl="id, name, country")
```

Stream Decorators

Stream decorators wrap other stream functions or perform operations on the stream. The are currently many stream decorators available: **complement**, **daemon**, **innerJoin**, **intersect**, **hashJoin**, **merge**, **leftOuterJoin**, **outerHashJoin**, **parallel**, **reduce**, **rollup**, **select**, **top**, **unique**, and **update.**

complement

The `complement` function wraps two streams (A and B) and emits tuples from A which do not exist in B. The tuples are emitted in the order in which they appear in stream A. Both streams must be sorted by the fields being used to determine equality (using the `on` parameter).

Parameters

- `StreamExpression for StreamA`
- `StreamExpression for StreamB`
- `on`: Fields to be used for checking equality of tuples between A and B. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName, otherFieldName=rightOtherFieldName"`.

Syntax

```
complement(
  search(collection1, q=a_s:(setA || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  search(collection1, q=a_s:(setB || setAB), fl="id,a_s,a_i", sort="a_i asc"),
  on="a_i"
)

complement(
  search(collection1, q=a_s:(setA || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  search(collection1, q=a_s:(setB || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  on="a_i,a_s"
)
```

daemon

The `daemon` function wraps another function and runs it at intervals using an internal thread. The daemon function can be used to provide both continuous push and pull streaming.

Continuous push streaming

With continuous push streaming the daemon function wraps another function and is then sent to the `/stream` handler for execution. The `/stream` handler recognizes the daemon function and keeps it resident in memory, so it can run it's internal function at intervals.

In order to facilitate the pushing of tuples, the daemon function must wrap another stream decorator that pushes the tuples somewhere. One example of this is the `update` function, which wraps a stream and sends the tuples to another SolrCloud collection for indexing.

Example:

```
daemon(id="uniqueId",
       runInterval="1000",
       update(destinationCollection,
              batchSize=100,
          topic(checkpointCollection,
                topicCollection,
                q="topic query",
                fl="id, title, abstract, text",
                id="topicId")
               )
        )
```

The sample code above shows a `daemon` function wrapping an `update` function, which is wrapping a `topic` function. When this expression is sent to the `/stream` handler, the `/stream` hander sees the daemon function and keeps it in memory where it will run at intervals. In this particular example, the daemon function will run the `update` function every second. The `update` function is wrapping a `topic` function, which returns all new documents for a specific query. The update function will send the new documents to another collection to be indexed.

The effect of this is to continuously push new documents that match a specific query into another collection. Custom push functions can be plugged in that push documents out of Solr and into other systems, such as Kafka or an email system.

Push streaming can also be used for continuous background aggregation scenarios where aggregates are rolled

up in the background at intervals and pushed to other Solr collections. Another use case is continuous background machine learning model optimization, where the optimized model is pushed to another Solr collection where they can be integrated into queries.

The `/stream` handler supports a small set commands for listing and controlling daemon functions:

```
http://localhost:8983/collection/stream?action=list
```

This command will provide a listing of the current daemon's running on the specific node along with there current state.

```
http://localhost:8983/collection/stream?action=stop&id=daemonId
```

This command will stop a specific daemon function but leave it resident in memory

```
http://localhost:8983/collection/stream?action=start&id=daemonId
```

This command will start a specific daemon function that has been stopped.

```
http://localhost:8983/collection/stream?action=kill&id=daemonId
```

This command will stop a specific daemon function and remove it from memory.

Continous Pull Streaming

The DaemonStream java class (part of the Solrj libraries) can also be embedded in a java application to provide continuous pull streaming. Sample code:

```
StreamContext context = new StreamContext()
SolrClientCache cache = new SolrClientCache();
context.setSolrClientCache(cache);

Map topicQueryParams = new HashMap();
topicQueryParams.put("q","hello");  // The query for the topic
topicQueryparams.put("rows", "500"); // How many rows to fetch during each run
topicQueryparams.put("fl", "id, "title"); // The field list to return with the
documents

TopicStream topicStream = new TopicStream(zkHost,        // Host address for the
zookeeper service housing the collections
                                          "checkpoints",  // The collection to store
the topic checkpoints
                                          "topicData",    // The collection to query
for the topic records
                                          "topicId",      // The id of the topic
                                          -1,             // checkpoint every X
tuples, if set -1 it will checkpoint after each run.
                                           topicQueryParams); // The query parameters
for the TopicStream

DaemonStream daemonStream = new DaemonStream(topicStream,             // The
underlying stream to run.
                                             "daemonId",              // The id of
the daemon
                                             1000,                    // The
```

```
interval at which to run the internal stream
                                                500);                    // The
internal queue size for the daemon stream. Tuples will be placed in the queue
                                                                           // as they are
read by the internal internal thread.
                                                                        // Calling read()
on the daemon stream reads records from the internal queue.

daemonStream.setStreamContext(context);

daemonStream.open();

//Read until it's time to shutdown the DaemonStream. You can define the shutdown
criteria.
while(!shutdown()) {
    Tuple tuple = daemonStream.read() // This will block until tuples become
available from the underlying stream (TopicStream)
                                      // The EOF tuple (signaling the end of the
stream) will never occur until the DaemonStream has been shutdown.
    //Do something with the tuples
}

// Shutdown the DaemonStream.
daemonStream.shutdown();

//Read the DaemonStream until the EOF Tuple is found.
//This allows the underlying stream to perform an orderly shutdown.

while(true) {
    Tuple tuple = daemonStream.read();
    if(tuple.EOF) {
  break;
    } else {
  //Do something with the tuples.
 }
```

```
}
//Finally close the stream
daemonStream.close();
```

leftOuterJoin

The `leftOuterJoin` function wraps two streams, Left and Right, and emits tuples from Left. If there is a tuple in Right equal (as defined by `on`) then the values in that tuple will be included in the emitted tuple. An equal tuple in Right **need not** exist for the Left tuple to be emitted. This supports one-to-one, one-to-many, many-to-one, and many-to-many left outer join scenarios. The tuples are emitted in the order in which they appear in the Left stream. Both streams must be sorted by the fields being used to determine equality (using the `on` parameter). If both tuples contain a field of the same name then the value from the Right stream will be used in the emitted tuple.

You can wrap the incoming streams with a `select` function to be specific about which field values are included in the emitted tuple.

Parameters

- `StreamExpression for StreamLeft`
- `StreamExpression for StreamRight`
- `on`: Fields to be used for checking equality of tuples between Left and Right. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName,otherFieldName=rightOtherFieldName"`.

Syntax

```
leftOuterJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  search(pets, q=type:cat, fl="personId,petName", sort="personId asc"),
  on="personId"
)

leftOuterJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  search(pets, q=type:cat, fl="ownerId,petName", sort="ownerId asc"),
  on="personId=ownerId"
)

leftOuterJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  select(
 search(pets, q=type:cat, fl="ownerId,name", sort="ownerId asc"),
 ownerId,
 name as petName
  ),
  on="personId=ownerId"
)
```

hashJoin

The `hashJoin` function wraps two streams, Left and Right, and for every tuple in Left which exists in Right will emit a tuple containing the fields of both tuples. This supports one-to-one, one-to-many, many-to-one, and many-to-many inner join scenarios. The tuples are emitted in the order in which they appear in the Left stream.

The order of the streams does not matter. If both tuples contain a field of the same name then the value from the Right stream will be used in the emitted tuple.

You can wrap the incoming streams with a `select` function to be specific about which field values are included in the emitted tuple.

The hashJoin function can be used when the tuples of Left and Right cannot be put in the same order. Because the tuples are out of order this stream functions by reading all values from the Right stream during the open operation and will store all tuples in memory. The result of this is a memory footprint equal to the size of the Right stream.

Parameters

- `StreamExpression for StreamLeft`
- `hashed=StreamExpression for StreamRight`
- on: Fields to be used for checking equality of tuples between Left and Right. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName, otherFieldName=rightOtherFieldName"`.

Syntax

```
hashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=search(pets, q=type:cat, fl="personId,petName", sort="personId asc"),
  on="personId"
)

hashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=search(pets, q=type:cat, fl="ownerId,petName", sort="ownerId asc"),
  on="personId=ownerId"
)

hashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=select(
 search(pets, q=type:cat, fl="ownerId,name", sort="ownerId asc"),
 ownerId,
 name as petName
  ),
  on="personId=ownerId"
)
```

innerJoin

Wraps two streams Left and Right and for every tuple in Left which exists in Right will emit a tuple containing the fields of both tuples. This supports one-one, one-many, many-one, and many-many inner join scenarios. The tuples are emitted in the order in which they appear in the Left stream. Both streams must be sorted by the fields being used to determine equality (the 'on' parameter). If both tuples contain a field of the same name then the value from the Right stream will be used in the emitted tuple. You can wrap the incoming streams with a select(...) to be specific about which field values are included in the emitted tuple.

Parameters

- `StreamExpression for StreamLeft`
- `StreamExpression for StreamRight`

- `on`: Fields to be used for checking equality of tuples between Left and Right. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName, otherFieldName=rightOtherFieldName"`.

Syntax

```
innerJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  search(pets, q=type:cat, fl="personId,petName", sort="personId asc"),
  on="personId"
)

innerJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  search(pets, q=type:cat, fl="ownerId,petName", sort="ownerId asc"),
  on="personId=ownerId"
)

innerJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  select(
 search(pets, q=type:cat, fl="ownerId,name", sort="ownerId asc"),
 ownerId,
 name as petName
  ),
  on="personId=ownerId"
)
```

intersect

The `intersect` function wraps two streams, A and B, and emits tuples from A which **DO** exist in B. The tuples are emitted in the order in which they appear in stream A. Both streams must be sorted by the fields being used to determine equality (the `on` parameter). Only tuples from A are emitted.

Parameters

- `StreamExpression for StreamA`
- `StreamExpression for StreamB`
- `on`: Fields to be used for checking equality of tuples between A and B. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName, otherFieldName=rightOtherFieldName"`.

Syntax

```
intersect(
  search(collection1, q=a_s:(setA || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  search(collection1, q=a_s:(setB || setAB), fl="id,a_s,a_i", sort="a_i asc"),
  on="a_i"
)

intersect(
  search(collection1, q=a_s:(setA || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  search(collection1, q=a_s:(setB || setAB), fl="id,a_s,a_i", sort="a_i asc, a_s
asc"),
  on="a_i,a_s"
)
```

merge

The `merge` function merges two or more streaming expressions and maintains the ordering of the underlying streams. Because the order is maintained, the sorts of the underlying streams must line up with the on parameter provided to the merge function.

Parameters

- `StreamExpression A`
- `StreamExpression B`
- `Optional StreamExpression C,D,....Z`
- `on`: Sort criteria for performing the merge. Of the form `fieldName order` where order is `asc` or `desc`. Multiple fields can be provided in the form `fieldA order, fieldB order`.

Syntax

```
# Merging two stream expressions together
merge(
      search(collection1,
             q="id:(0 3 4)",
             fl="id,a_s,a_i,a_f",
             sort="a_f asc"),
      search(collection1,
             q="id:(1)",
             fl="id,a_s,a_i,a_f",
             sort="a_f asc"),
      on="a_f asc")
```

```
# Merging four stream expressions together. Notice that while the sorts of each
stream are not identical they are
# comparable. That is to say the first N fields in each stream's sort matches the N
fields in the merge's on clause.
merge(
      search(collection1,
             q="id:(0 3 4)",
             fl="id,fieldA,fieldB,fieldC",
             sort="fieldA asc, fieldB desc"),
      search(collection1,
             q="id:(1)",
             fl="id,fieldA",
             sort="fieldA asc"),
      search(collection2,
             q="id:(10 11 13)",
             fl="id,fieldA,fieldC",
             sort="fieldA asc"),
      search(collection3,
             q="id:(987)",
             fl="id,fieldA,fieldC",
             sort="fieldA asc"),
      on="fieldA asc")
```

outerHashJoin

The `outerHashJoin` function wraps two streams, Left and Right, and emits tuples from Left. If there is a tuple in Right equal (as defined by the `on` parameter) then the values in that tuple will be included in the emitted tuple. An equal tuple in Right **need not** exist for the Left tuple to be emitted. This supports one-to-one, one-to-many, many-to-one, and many-to-many left outer join scenarios. The tuples are emitted in the order in which they appear in the Left stream. The order of the streams does not matter. If both tuples contain a field of the same name then the value from the Right stream will be used in the emitted tuple.

You can wrap the incoming streams with a `select` function to be specific about which field values are included in the emitted tuple.

The outerHashJoin stream can be used when the tuples of Left and Right cannot be put in the same order. Because the tuples are out of order, this stream functions by reading all values from the Right stream during the open operation and will store all tuples in memory. The result of this is a memory footprint equal to the size of the Right stream.

Parameters

- `StreamExpression for StreamLeft`
- `hashed=StreamExpression for StreamRight`
- `on`: Fields to be used for checking equality of tuples between Left and Right. Can be of the format `on="fieldName"`, `on="fieldNameInLeft=fieldNameInRight"`, or `on="fieldName,otherFieldName=rightOtherFieldName"`.

Syntax

```
outerHashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=search(pets, q=type:cat, fl="personId,petName", sort="personId asc"),
  on="personId"
)

outerHashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=search(pets, q=type:cat, fl="ownerId,petName", sort="ownerId asc"),
  on="personId=ownerId"
)

outerHashJoin(
  search(people, q=*:*, fl="personId,name", sort="personId asc"),
  hashed=select(
 search(pets, q=type:cat, fl="ownerId,name", sort="ownerId asc"),
 ownerId,
 name as petName
  ),
  on="personId=ownerId"
)
```

parallel

The `parallel` function wraps a streaming expression and sends it to N worker nodes to be processed in parallel.

The parallel function requires that the `partitionKeys` parameter be provided to the underlying searches. The `partitionKeys` parameter will partition the search results (tuples) across the worker nodes. Tuples with the same values in the partitionKeys field will be shuffled to the same worker nodes.

The parallel function maintains the sort order of the tuples returned by the worker nodes, so the sort criteria of the parallel function must match up with the sort order of the tuples returned by the workers.

Worker Collections
The worker nodes can be from the same collection as the data, or they can be a different collection entirely, even one that only exists for parallel streaming expressions. A worker collection can be any SolrCloud collection that has the `/stream` handler configured. Unlike normal SolrCloud collections, worker collections don't have to hold any data. Worker collections can be empty collections that exist only to execute streaming expressions.

Parameters

- `collection`: Name of the worker collection to send the StreamExpression to.
- `StreamExpression`: Expression to send to the worker collection.
- `workers`: Number of workers in the worker collection to send the expression to.
- `zkHost`: (Optional) The ZooKeeper connect string where the worker collection resides.
- `sort`: The sort criteria for ordering tuples returned by the worker nodes.

Syntax

```
parallel(workerCollection,
          reduce(
                 search(collection1, q=*:*, fl="id,a_s,a_i,a_f", sort="a_s desc",
partitionKeys="a_s"),
           by="a_s",
                 group(sort="a_f desc", n="4"))
          workers="20",
          zkHost="localhost:9983",
          sort="a_s desc")
```

The expression above shows a parallel function wrapping a reduce function. This will cause the reduce function to be run in parallel across 20 worker nodes.

reduce

The `reduce` function wraps an internal stream and groups tuples by common fields.

Each tuple group is operated on as a single block by a pluggable reduce operation. The group operation provided with Solr implements distributed grouping functionality. The group operation also serves as an example reduce operation that can be referred to when building custom reduce operations.



The reduce function relies on the sort order of the underlying stream. Accordingly the sort order of the underlying stream must be aligned with the group by field.

Parameters

- `StreamExpression`: (Mandatory)
- `by`: (Mandatory) A comma separated list of fields to group by.
- `Reduce Operation`: (Mandatory)

Syntax

```
reduce(
   search(collection1, q=*:*, fl="id,a_s,a_i,a_f", sort="a_s asc, a_f asc"),
   by="a_s",
   group(sort="a_f desc", n="4")
)
```

rollup

The `rollup` function wraps another stream function and rolls up aggregates over bucket fields. The rollup function relies on the sort order of the underlying stream to rollup aggregates one grouping at a time. Accordingly, the sort order of the underlying stream must match the fields in the `over` parameter of the rollup function.

The rollup function also needs to process entire result sets in order to perform it's aggregations. When the underlying stream is the `search` function, the `/export` handler can be used to provide full sorted result sets to the rollup function. This sorted approach allows the rollup function to perform aggregations over very high cardinality fields. The disadvantage of this approach is that the tuples must be sorted and streamed across the network to a worker node to be aggregated. For faster aggregation over low to moderate cardinality fields, the `facet` function can be used.

Parameters

- `StreamExpression` (Mandatory)
- `over`: (Mandatory) A list of fields to group by.
- `metrics`: (Mandatory) The list of metrics to compute. Currently supported metrics are `sum(col)`, `avg(col)`, `min(col)`, `max(col)`, `count(*)`.

Syntax

```
rollup(
   search(
      collection1, q=*:*, fl="a_s,a_i,a_f", qt="/export", sort="a_s asc"),
   over="a_s",
   sum(a_i),
   sum(a_f),
   min(a_i),
   min(a_f),
   max(a_i),
   max(a_f),
   avg(a_i),
   avg(a_f),
   count(*)
)
```

The example about shows the rollup function wrapping the search function. Notice that search function is using the `/export` handler to provide the entire result set to the rollup stream. Also notice that the search function's **sort param** matches up with the rollup's `over` parameter. This allows the rollup function to rollup the over the `a_s` field, one group at a time.

scoreNodes

See section in graph traversal.

select

The `select` function wraps a streaming expression and outputs tuples containing a subset or modified set of fields from the incoming tuples. The list of fields included in the output tuple can contain aliases to effectively rename fields. One can provide a list of operations to perform on any fields, such as `replace` to replace the value of a field with some other value or the value of another field in the tuple.

Parameters

- `StreamExpression`
- `fieldName`: name of field to include in the output tuple (can include multiple of these) outputTuple[fieldName] = inputTuple[fieldName]
- `fieldName as aliasFieldName`: aliased field name to include in the output tuple (can include multiple of these) outputTuple[aliasFieldName] = incomingTuple[fieldName]
- `replace(fieldName, value, withValue=replacementValue)`: if incomingTuple[fieldName] == value then outgoingTuple[fieldName] will be set to replacementValue. value can be the string "null" to replace a null value with some other value
- `replace(fieldName, value, withField=otherFieldName)`: if incomingTuple[fieldName] == value then outgoingTuple[fieldName] will be set to the value of incomingTuple[otherFieldName]. value can be the string "null" to replace a null value with some other value

Syntax

```
// output tuples with fields teamName, wins, and losses where a null value for wins
or losses is translated to the value of 0
select(
  search(collection1, fl="id,teamName_s,wins,losses", q="*:*", sort="id asc"),
  teamName_s as teamName,
  wins,
  losses,
  replace(wins,null,withValue=0),
  replace(losses,null,withValue=0)
)
```

sort

The `sort` function wraps a streaming expression and re-orders the tuples. The sort function emits all incoming tuples in the new sort order. The sort function reads all tuples from the incoming stream, re-orders them using an algorithm with `O(nlog(n))` performance characteristics, where n is the total number of tuples in the incoming stream, and then outputs the tuples in the new sort order. Because all tuples are read into memory, the memory consumption of this function grows linearly with the number of tuples in the incoming stream.

Parameters

- `StreamExpression`
- `by`: Sort criteria for re-ordering the tuples

Syntax

The expression below finds dog owners and orders the results by owner and pet name. Notice that it uses an efficient innerJoin by first ordering by the person/owner id and then re-orders the final output by the owner and pet names.

```
sort(
  innerJoin(
    search(people, q=*:*, fl="id,name", sort="id asc"),
    search(pets, q=type:dog, fl="owner,petName", sort="owner asc"),
    on="id=owner"
  ),
  by="name asc, petName asc"
)
```

top

The `top` function wraps a streaming expression and re-orders the tuples. The top function emits only the top N tuples in the new sort order. The top function re-orders the underlying stream so the sort criteria **does not** have to match up with the underlying stream.

Parameters

- `n`: Number of top tuples to return.
- `StreamExpression`
- `sort`: Sort criteria for selecting the top N tuples.

Syntax

The expression below finds the top 3 results of the underlying search. Notice that it reverses the sort order. The top function re-orders the results of the underlying stream.

```
top(n=3,
     search(collection1,
            q="*:*",
            qt="/export",
            fl="id,a_s,a_i,a_f",
            sort="a_f desc, a_i desc"),
      sort="a_f asc, a_i asc")
```

unique

The `unique` function wraps a streaming expression and emits a unique stream of tuples based on the `over` par ameter. The unique function relies on the sort order of the underlying stream. The `over` parameter must match up with the sort order of the underlying stream.

The unique function implements a non-co-located unique algorithm. This means that records with the same unique `over` field do not need to be co-located on the same shard. When executed in the parallel, the `partiti onKeys` parameter must be the same as the unique `over` field so that records with the same keys will be shuffled to the same worker.

Parameters

- `StreamExpression`
- `over`: The unique criteria.

Syntax

```
unique(
  search(collection1,
         q="*:*",
         qt="/export",
         fl="id,a_s,a_i,a_f",
         sort="a_f asc, a_i asc"),
  over="a_f")
```

update

The `update` function wraps another functions and sends the tuples to a SolrCloud collection for indexing.

Parameters

- `destinationCollection`: (Mandatory) The collection where the tuples will indexed.
- `batchSize`: (Mandatory) The indexing batch size.
- `StreamExpression`: (Mandatory)

Syntax

```
update(destinationCollection,
       batchSize=500,
       search(collection1,
              q=*:*,
              fl="id,a_s,a_i,a_f,s_multi,i_multi",
              sort="a_f asc, a_i asc"))
```

The example above sends the tuples returned by the `search` function to the `destinationCollection` to be indexed.

Graph Traversal

Graph traversal with streaming expressions uses the `gatherNodes` function to perform a breadth-first graph traversal. The `gatherNodes` function can be combined with the `scoreNodes` function to provide recommendations. `gatherNodes` can also be combined with the wider streaming expression library to perform complex operations on gathered node sets.

`gatherNodes` traversals are distributed within a SolrCloud collection and can span collections.

`gatherNodes` is designed for use cases that involve zooming into a neighborhood in the graph and performing precise traversals to gather node sets and aggregations. In these types of use cases `gatherNodes` will often provide sub-second performance. Some sample use cases are provided later in the document.

- Basic Syntax
- Aggregations
- Nesting gatherNodes functions
- Cycle Detection
- Filtering the Traversal
- Root Streams
- Skipping High Frequency Nodes
- Tracking the Traversal
- Cross-Collection Traversals
- Combining gatherNodes With Other Streaming Expressions
- Sample Use Cases
  - Calculate Market Basket Co-occurrence
  - Using the scoreNodes Function to Make a Recommendation
  - Recommend Content Based on Collaborative Filter
  - Protein Pathway Traversal
- Exporting GraphML to Support Graph Visualization
  - Sample Request
  - Sample GraphML Output



This document assumes a basic understanding of graph terminology and streaming expressions. You can begin exploring graph traversal concepts with this Wikipedia article. More details about streaming expressions are available in this Guide, in the section Streaming Expressions.

Basic Syntax

We'll start with the most basic syntax and slowly build up more complexity. The most basic syntax for `gatherNodes` is:

```
gatherNodes(emails,
            walk="johndoe@apache.org->from",
            gather="to")
```

Let's break down this simple expression.

The first parameter, `emails`, is the collection being traversed. The second parameter, `walk`, maps a hard-coded node ID ("johndoe@apache.org") to a field in the index (`from`). This will return all the **edges** in the index that have `johndoe@apache.org` in the `from` field.

The `gather` parameter tells the function to gather the values in the `to` field. The values that are gathered are the node IDs emitted by the function.

In the example above the nodes emitted will be all of the people that "johndoe@apache.org" has emailed.

The walk parameter also accepts a list of root node IDs:

```
gatherNodes(emails,
            walk="johndoe@apache.org, janesmith@apache.org->from",
            gather="to")
```

The `gatherNodes` function above finds all the edges with "johndoe@apache.org" or "janesmith@apache.org" in the `from` field and gathers the `to` field.

Like all Streaming Expressions, you can execute a `gatherNodes` expression by sending it to the `/stream` handler. For example:

```
curl --data-urlencode 'expr=gatherNodes(emails,
                                        walk="johndoe@apache.org,
janesmith@apache.org->from",
                                        gather="to")'
http://localhost:8983/solr/emails/stream
```

The output of this expression would look like this:

```
{
  "result-set": {
    "docs": [
      {
        "node": "slist@campbell.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "node": "catherine.pernot@enron.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "node": "airam.arteaga@enron.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "EOF": true,
        "RESPONSE_TIME": 44
      }
    ]
  }
}
```

All of the tuples returned have the `node` field. The `node` field contains the node IDs gathered by the function. The `collection`, `field`, and `level` of the traversal are also included in the output.

Notice that the level is "1" for each tuple in the example. The root nodes are level 0 (in the example above, the root nodes are "johndoe@apache.org, janesmith@apache.org") By default the `gatherNodes` function emits only the ***leaf nodes*** of the traversal, which is the outer-most node set. To emit the root nodes you can specify the `scatter` parameter:

```
gatherNodes(emails,
            walk="johndoe@apache.org->from",
            gather="to",
            scatter="branches, leaves")
```

The `scatter` parameter controls whether to emit the ***branches*** with the ***leaves***. The root nodes are considered "branches" because they are not the outer-most level of the traversal.

When scattering both branches and leaves the output would like this:

```
{
  "result-set": {
    "docs": [
      {
        "node": "johndoe@apache.org",
        "collection": "emails",
        "field": "node",
        "level": 0
      },
      {
        "node": "slist@campbell.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "node": "catherine.pernot@enron.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "node": "airam.arteaga@enron.com",
        "collection": "emails",
        "field": "to",
        "level": 1
      },
      {
        "EOF": true,
        "RESPONSE_TIME": 44
      }
    ]
  }
}
```

Now the level 0 root node is included in the output.

Aggregations

`gatherNodes` also supports aggregations. For example:

```
gatherNodes(emails,
            walk="johndoe@apache.org, janesmith@apache.org->from",
            gather="to",
            count(*))
```

The expression above finds the edges with "johndoe@apache.org" or "janesmith@apache.org" in the `from` field and gathers the values from the `to` field. It also aggregates the count for each node ID gathered.

A gathered node could have a count of 2 if both "johndoe@apache.org" and "janesmith@apache.org" have emailed the same person. Node sets contain a unique set of nodes, so the same person won't appear twice in the node set, but the count will reflect that it appeared twice during the traversal.

Edges are uniqued as part of the traversal so the count will **not** reflect the number of times "johndoe@apache.org" emailed the same person. For example, personA might have emailed personB 100 times. These edges would get uniqued and only be counted once. But if person personC also emailed personB

this would increment the count for personB.

The aggregation functions supported are `count(*)`, `sum(field)`, `min(field)`, `max(field)`, and `avg(field)`. The fields being aggregated should be present in the edges collected during the traversal. Later examples (below) will show aggregations can be a powerful tool for providing recommendations and limiting the scope of traversals.

Nesting gatherNodes functions

The `gatherNodes` function can be nested to traverse deeper into the graph. For example:

```
gatherNodes(emails,
            gatherNodes(emails,
                        walk="johndoe@apache.org->from",
                        gather="to"),
            walk="node->from",
            gather="to")
```

In the example above the outer `gatherNodes` function operates on the node set collected from the inner `gatherNodes` function.

Notice that the inner `gatherNodes` function behaves exactly as the examples already discussed. But the `walk` parameter of the outer `gatherNodes` function behaves differently.

In the outer `gatherNodes` function the `walk` parameter works with tuples coming from an internal streaming expression. In this scenario the `walk` parameter maps the `node` field to the `from` field. Remember that the node IDs collected from the inner `gatherNodes` expression are placed in the `node` field.

Put more simply, the inner expression gathers all the people that "johndoe@apache.org" has emailed. We can call this group the "friends of johndoe@apache.org". The outer expression gathers all the people that the "friends of johndoe@apache.org" have emailed. This is a basic friends-of-friends traversal.

This construct of nesting `gatherNodes` functions is the basic technique for doing a controlled traversal through the graph.

Cycle Detection

The `gatherNodes` function performs cycle detection across the entire traversal. This ensures that nodes that have already been visited are not traversed again. Cycle detection is important for both limiting the size of traversals and gathering accurate aggregations. Without cycle detection the size of the traversal could grow exponentially with each hop in the traversal. With cycle detection only new nodes encountered are traversed.

Cycle detection **does not** cross collection boundaries. This is because internally the collection name is part of the node ID. For example the node ID "johndoe@apache.org", is really `emails/johndoe@apache.org`. When traversing to another collection "johndoe@apache.org" will be traversed.

Filtering the Traversal

Each level in the traversal can be filtered with a filter query. For example:

```
gatherNodes(emails,
            walk="johndoe@apache.org->from",
            fq="body:(solr rocks)",
            gather="to")
```

In the example above only emails that match the filter query will be included in the traversal. Any Solr query can be included here. So you can do fun things like geospatial queries, apply any of the available query parsers, or even write custom query parsers to limit the traversal.

Root Streams

Any streaming expression can be used to provide the root nodes for a traversal. For example:

```
gatherNodes(emails,
            search(emails, q="body:(solr rocks)", fl="to", sort="score desc",
rows="20")
            walk="to->from",
            gather="to")
```

The example above provides the root nodes through a search expression. You can also provide arbitrarily complex, nested streaming expressions with joins, etc., to specify the root nodes.

Notice that the `walk` parameter maps a field from the tuples generated by the inner stream. In this case it maps the `to` field from the inner stream to the `from` field.

Skipping High Frequency Nodes

It's often desirable to skip traversing high frequency nodes in the graph. This is similar in nature to a search term stop list. The best way to describe this is through an example use case.

Let's say that you want to recommend content for a user based on a collaborative filter. Below is one approach for a simple collaborative filter:

1. Find all content userA has read.
2. Find users whose reading list is closest to userA. These are users with similar tastes as userA.
3. Recommend content based on what the users in step 2 have read, that userA has not yet read.

Look closely at step 2. In large graphs, step 2 can lead to a very large traversal. This is because userA may have viewed content that has been viewed by millions of other people. We may want to skip these high frequency nodes for two reasons:

1. A large traversal that visit millions of unique nodes is slow and takes a lot of memory because cycle detection is tracked in memory.
2. High frequency nodes are also not useful in determining users with similar tastes. The content that fewer people have viewed provides a more precise recommendation.

The `gatherNodes` function has the `maxDocFreq` param to allow for filtering out high frequency nodes. The sample code below shows steps 1 and 2 of the recommendation:

```
gatherNodes(logs,
             search(logs, q="userID:user1", fl="articleID", sort="articleID asc",
fq="action:view", qt="/export"),
             walk="articleID->articleID",
             gather="userID",
             fq="action:view",
             maxDocFreq="10000",
             count(*)))
```

In the example above, the inner search expression searches the `logs` collection and returning all the articles viewed by "user1". The outer `gatherNodes` expression takes all the articles emitted from the inner search expression and finds all the records in the logs collection for those articles. It then gathers and aggregates the users that have read the articles. The `maxDocFreq` parameter limits the articles returned to those that appear in

no more then 10,000 log records (per shard). This guards against returning articles that have been viewed by millions of users.

Tracking the Traversal

By default the `gatherNodes` function only tracks enough information to do cycle detection. This provides enough information to output the nodes and aggregations in the graph.

For some use cases, such as graph visualization, we also need to output the edges. Setting `trackTraversal="true"` tells `gatherNodes` to track the connections between nodes, so the edges can be constructed. When `trackTraversal` is enabled a new `ancestors` property will appear with each node. The `ancestors` property contains a list of node IDs that pointed to the node.

Below is a sample `gatherNodes` expression with `trackTraversal` set to true:

```
gatherNodes(emails,
            gatherNodes(emails,
                        walk="johndoe@apache.org->from",
                        gather="to",
                        trackTraversal="true"),
            walk="node->from",
            trackTraversal="true",
            gather="to")
```

Cross-Collection Traversals

Nested `gatherNodes` functions can operate on different SolrCloud collections. This allow traversals to "walk" from one collection to another to gather nodes. Cycle detection does not cross collection boundaries, so nodes collected in one collection will be traversed in a different collection. This was done deliberately to support cross-collection traversals. Note that the output from a cross-collection traversal will likely contain duplicate nodes with different collection attributes.

Below is a sample `gatherNodes` expression that traverses from the "emails" collection to the "logs" collection:

```
gatherNodes(logs,
            gatherNodes(emails,
               search(emails, q="body:(solr rocks)", fl="from", sort="score desc",
rows="20")
                        walk="from->from",
                        gather="to",
                        scatter="leaves, branches"),
            walk="node->user",
            fq="action:edit",
            gather="contentID")
```

The example above finds all people who sent emails with a body that contains "solr rocks". It then finds all the people these people have emailed. Then it traverses to the logs collection and gathers all the content IDs that these people have edited.

Combining gatherNodes With Other Streaming Expressions

The `gatherNodes` function can act as both a stream source and a stream decorator. The connection with the wider stream expression library provides tremendous power and flexibility when performing graph traversals. Here is an example of using the streaming expression library to intersect two friend networks:

```
intersect(on="node",
                     sort(by="node asc",
                          gatherNodes(emails,
                                      gatherNodes(emails,
                                                  walk="johndoe@apache.org->from",
                                                  gather="to"),
                                      walk="node->from",
                                      gather="to",
                                      scatter="branches,leaves")),
                      sort(by="node asc",
                           gatherNodes(emails,
                                       gatherNodes(emails,
                                                   walk="janedoe@apache.org->from",

                                                   gather="to"),
                                       walk="node->from",
                                       gather="to",
                                       scatter="branches,leaves")))
```

The example above gathers two separate friend networks, one rooted with "johndoe@apache.org" and another rooted with "janedoe@apache.org". The friend networks are then sorted by the `node` field, and intersected. The resulting node set will be the intersection of the two friend networks.

Sample Use Cases

Calculate Market Basket Co-occurrence

It is often useful to know which products are most frequently purchased with a particular product. This example uses a simple market basket table (indexed in Solr) to store past shopping baskets. The schema for the table is very simple with each row containing a `basketID` and a `productID`. This can be seen as a graph with each row in the table representing an edge. And it can be traversed very quickly to calculate basket co-occurrence, even when the graph contains billions of edges.

Here is the sample syntax:

```
top(n="5",
    sort="count(*) desc",
    gatherNodes(baskets,
                random(baskets, q="productID:ABC", fl="basketID", rows="500"),
                walk="basketID->basketID",
                fq="-productID:ABC",
                gather="productID",
                count(*)))
```

Let's break down exactly what this traversal is doing.

1. The first expression evaluated is the inner `random` expression, which returns 500 random basketIDs, from the `baskets` collection, that have the `productID` "ABC". The `random` expression is very useful for recommendations because it limits the traversal to a fixed set of baskets, and because it adds the element of surprise into the recommendation. Using the `random` function you can provide fast sample sets from very large graphs.
2. The outer `gatherNodes` expression finds all the records in the `baskets` collection for the basketIDs generated in step 1. It also filters out `productID` "ABC" so it doesn't show up in the results. It then gathers and counts the productID's across these baskets.
3. The outer `top` expression ranks the productIDs emitted in step 2 by the count and selects the top 5.

In a nutshell this expression finds the products that most frequently co-occur with product "ABC" in past shopping baskets.

Using the scoreNodes Function to Make a Recommendation

This use case builds on the market basket example above that calculates which products co-occur most frequently with productID:ABC. The ranked co-occurrence counts provide candidates for a recommendation. The `scoreNodes` function can be used to score the candidates to find the best recommendation.

Before diving into the syntax of the `scoreNodes` function it's useful to understand why the raw co-occurrence counts may not produce the best recommendation. The reason is that raw co-occurrence counts favor items that occur frequently across all baskets. A better recommendation would find the product that has the most significant relationship with productID ABC. The scoreNodes functions uses a term frequency-inverse document frequency (TF-IDF) algorithm to find the most significant relationship.

How It Works

The scoreNodes function assigns a score to each node emitted by the gatherNodes expression. By default the scoreNodes function uses the `count(*)` aggregation, which is the co-occurrence count, as the TF value. The IDF value for each node is fetched from the collection where the node was gathered. Each node is then scored using the TF*IDF formula, which provides a boost to nodes with a lower frequency across all market baskets.

Combining the co-occurrence count with the IDF provides a score that shows how important the relationship is between productID ABC and the recommendation candidates.

The scoreNodes function adds the score to each node in the `nodeScore` field.

Example Syntax

```
top(n="1",
    sort="nodeScore desc",
    scoreNodes(top(n="50",
                   sort="count(*) desc",
                   gatherNodes(baskets,
                               random(baskets, q="productID:ABC", fl="basketID",
rows="500"),
                               walk="basketID->basketID",
                               fq="-productID:ABC",
                               gather="productID",
                               count(*)))))
```

This example builds on the earlier example "Calculate market basket co-occurrence".

1. Notice that the inner-most top function is taking the top 50 products that co-occur most frequently with productID ABC. This provides 50 candidate recommendations.
2. The `scoreNodes` function then assigns a score to the candidates based on the TF*IDF of each node.
3. The outer `top` expression selects the highest scoring node. This is the recommendation.

Recommend Content Based on Collaborative Filter

In this example we'll recommend content for a user based on a collaborative filter. This recommendation is made using log records that contain the `userID` and `articleID` and the action performed. In this scenario each log record can be viewed as an edge in a graph. The userID and articleID are the nodes and the action is an edge property used to filter the traversal.

Here is the sample syntax:

```
top(n="5",
    sort="count(*) desc",
    gatherNodes(logs,
                top(n="30",
                    sort="count(*) desc",
                    gatherNodes(logs,
                                search(logs, q="userID:user1", fl="articleID",
sort="articleID asc", fq="action:read", qt="/export"),
                                walk="articleID->articleID",
                                gather="userID",
                                fq="action:read",
                                maxDocFreq="10000",
                                count(*))),
                walk="node->userID",
                gather="articleID",
                fq="action:read",
                count(*)))
```

Let's break down the expression above step-by-step.

1. The first expression evaluated is the inner `search` expression. This expression searches the `logs` collection for all records matching "user1". This is the user we are making the recommendation for. There is a filter applied to pull back only records where the "action:read". It returns the `articleID` for each record found. In other words, this expression returns all the articles "user1" has read.
2. The inner `gatherNodes` expression operates over the articleIDs returned from step 1. It takes each `articleID` found and searches them against the `articleID` field. Note that it skips high frequency nodes using the `maxDocFreq` param to filter out articles that appear over 10,000 times in the logs. It gathers userIDs and aggregates the counts for each user. This step finds the users that have read the same articles that "user1" has read and counts how many of the same articles they have read.
3. The inner `top` expression ranks the users emitted from step 2. It will emit the top 30 users who have the most overlap with user1's reading list.
4. The outer `gatherNodes` expression gathers the reading list for the users emitted from step 3. It counts the articleIDs that are gathered. Any article selected in step 1 (user1 reading list), will not appear in this step due to cycle detection. So this step returns the articles read by the users with the most similar readings habits to "user1" that "user1" has not read yet. It also counts the number of times each article has been read across this user group.
5. The outer `top` expression takes the top articles emitted from step 4. This is the recommendation.

Protein Pathway Traversal

In recent years, scientists have become increasingly able to rationally design drugs that target the mutated proteins, called oncogenes, responsible for some cancers. Proteins typically act through long chains of chemical interactions between multiple proteins, called pathways, and, while the oncogene in the pathway may not have a corresponding drug, another protein in the pathway may. Graph traversal on a protein collection that records protein interactions and drugs may yield possible candidates. (Thanks to Lewis Geer of the NCBI, for providing this example).

The example below illustrates a protein pathway traversal:

```
gatherNodes(proteins,
            gatherNodes(proteins,
                        walk="NRAS->name",
                        gather="interacts"),
            walk="node->name",
            gather="drug")
```

Let's break down exactly what this traversal is doing.

1. The inner `gatherNodes` expression traverses in the `proteins` collection. It finds all the edges in the graph where the name of the protein is "NRAS". Then it gathers the proteins in the `interacts` field. This gathers all the proteins that "NRAS" interactions with.
2. The outer `gatherNodes` expression also works with the `proteins` collection. It gathers all the drugs that correspond to proteins emitted from step 1.
3. Using this stepwise approach you can gather the drugs along the pathway of interactions any number of steps away from the root protein.

Exporting GraphML to Support Graph Visualization

In the examples above, the `gatherNodes` expression was sent to Solr's `/stream` handler like any other streaming expression. This approach outputs the nodes in the same JSON tuple format as other streaming expressions so that it can be treated like any other streaming expression. You can use the `/stream` handler when you need to operate directly on the tuples, such as in the recommendation use cases above.

There are other graph traversal use cases that involve graph visualization. Solr supports these use cases with the introduction of the `/graph` request handler, which takes a `gatherNodes` expression and outputs the results in GraphML.

GraphML is an XML format supported by graph visualization tools such as Gephi, which is a sophisticated open source tool for statistically analyzing and visualizing graphs. Using a `gatherNodes` expression, parts of a larger graph can be exported in GraphML and then imported into tools like Gephi.

There are a few things to keep mind when exporting a graph in GraphML

1. The `/graph` handler can export both the nodes and edges in the graph. By default, it only exports the nodes. To export the edges you must set `trackTraversal="true"` in the `gatherNodes` expression.
2. The `/graph` handler currently accepts an arbitrarily complex streaming expression which includes a `gatherNodes` expression. If the streaming expression doesn't include a `gatherNodes` expression, the `/graph` handler will not properly output GraphML.
3. The `/graph` handler currently accepts a single arbitrarily complex, nested `gatherNodes` expression per request. This means you cannot send in a streaming expression that joins or intersects the node sets from multiple `gatherNodes` expressions. The `/graph` handler does support any level of nesting within a single `gatherNodes` expression. The `/stream` handler does support joining and intersecting node sets, but the `/graph` handler currently does not.

Sample Request

```
curl --data-urlencode 'expr=gatherNodes(enron_emails,
                                        gatherNodes(enron_emails,

walk="kayne.coulter@enron.com->from",
                                                    trackTraversal="true",
                                                    gather="to"),
                                        walk="node->from",
                                        scatter="leaves,branches",
                                        trackTraversal="true",
                                        gather="to")'
http://localhost:8983/solr/enron_emails/graph
```

Sample GraphML Output

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
<graph id="G" edgedefault="directed">
     <node id="kayne.coulter@enron.com">
           <data key="field">node</data>
           <data key="level">0</data>
           <data key="count(*)">0.0</data>
     </node>
     <node id="don.baughman@enron.com">
           <data key="field">to</data>
           <data key="level">1</data>
           <data key="count(*)">1.0</data>
     </node>
     <edge id="1"  source="kayne.coulter@enron.com"
target="don.baughman@enron.com"/>
     <node id="john.kinser@enron.com">
           <data key="field">to</data>
           <data key="level">1</data>
           <data key="count(*)">1.0</data>
    </node>
    <edge id="2"  source="kayne.coulter@enron.com"  target="john.kinser@enron.com"/>
    <node id="jay.wills@enron.com">
          <data key="field">to</data>
          <data key="level">1</data>
          <data key="count(*)">1.0</data>
    </node>
    <edge id="3"  source="kayne.coulter@enron.com"  target="jay.wills@enron.com"/>
</graph></graphml>
```

Parallel SQL Interface

Solr's Parallel SQL Interface brings the power of SQL to SolrCloud. The SQL interface seamlessly combines SQL with Solr's full-text search capabilities. Two implementations for aggregations allow using either MapReduce-like shuffling or the JSON Facet API, depending on performance needs. These features allow Solr's SQL interface to be used for a wide variety of use cases.

- SQL Architecture
  - Solr Collections and DB Tables
  - Aggregation Modes
- Configuration
  - /sql Request Handler
  - /stream and /export Request Handlers
  - Fields
- Sending Queries
  - JDBC Driver
  - HTTP Interface
- Solr SQL Syntax
  - SELECT Statements
    - Basic SELECT statement with LIMIT
    - WHERE Clause and Boolean Predicates
    - ORDER BY Clause

        - LIMIT Clause
        - SELECT DISTINCT Queries
    - Statistics
    - GROUP BY Aggregations
        - Basic GROUP BY with Aggregates
        - The Column Identifiers and Aliases
        - GROUP BY Clause
        - HAVING Clause
        - ORDER BY Clause
- Best Practices
    - Separate Collections
- Parallel SQL Queries
    - Parallelized Queries
        - SQL Tier
        - Worker Tier
        - Data Table Tier
- SQL Clients and Database Visualization Tools
    - Generic
    - DbVisualizer
    - SQuirreL SQL
    - Apache Zeppelin (incubating)

SQL Architecture

The SQL interface allows sending a SQL query to Solr and getting documents streamed back in response. Under the covers, Solr's SQL interface is powered by the Presto Project's SQL Parser, which translates SQL queries on the fly to Streaming Expressions.

Solr Collections and DB Tables

In a standard `SELECT` statement such as "`SELECT <expressions> FROM <table>`", the table names correspond to Solr collection names. Table names are case insensitive.

Column names in the SQL query map directly to fields in the Solr index for the collection being queried. These identifiers are case sensitive. Aliases are supported, and can be referenced in the `ORDER BY` clause.

The * syntax to indicate all fields is not supported in either limited or unlimited queries. The `score` field can be used only with queries that contain a `LIMIT` clause.

For example, we could index Solr's sample documents and then construct an SQL query like this:

```
SELECT manu as mfr, price as retail FROM techproducts
```

The collection in Solr we are using is "techproducts", and we've asked for the "manu" and "price" fields to be returned and aliased with new names. While this example does not use those aliases, we could build on this to ORDER BY one or more of those fields.

More information about how to structure SQL queries for Solr is included in the section Solr SQL Syntax.

Aggregation Modes

The SQL feature of Solr can work with aggregations (grouping of results) in two ways:

- `map_reduce`: This implementation shuffles tuples to worker nodes and performs the aggregation on the worker nodes. It involves sorting and partitioning the entire result set and sending it to worker nodes. In this approach the tuples arrive at the worker nodes sorted by the GROUP BY fields. The worker nodes

can then rollup the aggregates one group at a time. This allows for unlimited cardinality aggregation, but you pay the price of sending the entire result set across the network to worker nodes.
- `facet`: This uses the JSON Facet API or StatsComponent for aggregations. In this scenario the aggregations logic is pushed down into the search engine and only the aggregates are sent across the network. This is Solr's normal mode of operation. This is fast when the cardinality of GROUP BY fields is low to moderate. But it breaks down when you have high cardinality fields in the GROUP BY field.

These modes are defined with the `aggregationMode` property when sending the request to Solr.

As noted, the choice between aggregation modes depends on the cardinality of the fields you are working with. If you have low-to-moderate cardinality in the fields you are grouping by, the 'facet' aggregation mode will give you a higher performance because only the final groups are returned, very similar to how facets work today. If, however, you have high cardinality in the fields, the "map_reduce" aggregation mode with worker nodes provide a much more performant option.

More detail on the architecture of the "map_reduce" query is in the section Parallel Query Architecture.

Configuration

The request handlers used for the SQL interface are configured to load implicitly, meaning there is little to do to start using this feature.

/sql Request Handler

The `/sql` handler is the front end of the Parallel SQL interface. All SQL queries are sent to the `/sql` handler to be processed. The handler also coordinates the distributed MapReduce jobs when running `GROUP BY` and `SELECT DISTINCT` queries in `map_reduce` mode. By default the `/sql` handler will choose worker nodes from it's own collection to handle the distributed operations. In this default scenario the collection where the `/sql` handler resides acts as the default worker collection for MapReduce queries.

By default, the `/sql` request handler is configured as an implicit handler, meaning that it is always enabled in every Solr installation and no further configuration is required.



As described below in the section Best Practices, you may want to set up a separate collection for parallelized SQL queries. If you have high cardinality fields and a large amount of data, please be sure to review that section and

/stream and /export Request Handlers

The Streaming API is an extensible parallel computing framework for SolrCloud. Streaming Expressions provide a query language and a serialization format for the Streaming API. The Streaming API provides support for fast MapReduce allowing it to perform parallel relational algebra on extremely large data sets. Under the covers the SQL interface parses SQL queries using the Presto SQL Parser. It then translates the queries to the parallel query plan. The parallel query plan is expressed using the Streaming API and Streaming Expressions.

Like the `/sql` request handler, the `/stream` and `/export` request handlers are configured as implicit handlers, and no further configuration is required.

Fields

In some cases, fields used in SQL queries must be configured as DocValue fields. If queries are unlimited, all fields must be DocValue fields. If queries are limited (with the `limit` clause) then fields do not have to be have DocValues enabled.

Sending Queries

The SQL Interface provides a basic JDBC driver and an HTTP interface to perform queries.

JDBC Driver

The JDBC Driver ships with SolrJ. Below is sample code for creating a connection and executing a query with the JDBC driver:

```
Connection con = null;
try {
 con = DriverManager.getConnection("jdbc:solr://" + zkHost +
"?collection=collection1&aggregationMode=map_reduce&numWorkers=2");
 stmt = con.createStatement();
 rs = stmt.executeQuery("SELECT a_s, sum(a_f) as sum FROM collection1 GROUP BY a_s
ORDER BY sum desc");

 while(rs.next()) {
  String a_s = rs.getString("a_s");
  double s = rs.getDouble("sum");
 }
} finally {
 rs.close();
 stmt.close();
 con.close();
}
```

The connection URL must contain the `zkHost` and the `collection` parameters. The collection must be a valid SolrCloud collection at the specified ZooKeeper host. The collection must also be configured with the `/sql` handler. The `aggregationMode` and `numWorkers` parameters are optional.

HTTP Interface

Solr accepts parallel SQL queries through the `/sql` handler.

Below is a sample curl command performing a SQL aggregate query in facet mode:

```
curl --data-urlencode 'stmt=SELECT to, count(*) FROM collection4 GROUP BY to ORDER
BY count(*) desc LIMIT 10'
   http://localhost:8983/solr/collection4/sql?aggregationMode=facet
```

Below is sample result set:

```
{"result-set":{"docs":[
   {"count(*)":9158,"to":"pete.davis@enron.com"},
   {"count(*)":6244,"to":"tana.jones@enron.com"},
   {"count(*)":5874,"to":"jeff.dasovich@enron.com"},
   {"count(*)":5867,"to":"sara.shackleton@enron.com"},
   {"count(*)":5595,"to":"steven.kean@enron.com"},
   {"count(*)":4904,"to":"vkaminski@aol.com"},
   {"count(*)":4622,"to":"mark.taylor@enron.com"},
   {"count(*)":3819,"to":"kay.mann@enron.com"},
   {"count(*)":3678,"to":"richard.shapiro@enron.com"},
   {"count(*)":3653,"to":"kate.symes@enron.com"},
   {"EOF":"true","RESPONSE_TIME":10}]}
}
```

Notice that the result set is an array of tuples with key/value pairs that match the SQL column list. The final tuple contains the EOF flag which signals the end of the stream.

Solr SQL Syntax

Solr supports a broad range of SQL syntax.

SQL Parser is Case Insensitive
The SQL parser being used by Solr to translate the SQL statements is case insensitive. However, for ease of reading, all examples on this page use capitalized keywords.

SELECT Statements

Solr supports limited and unlimited select queries. The syntax between the two types of queries are identical except for the `LIMIT` clause in the SQL statement. However, they have very different execution plans and different requirements for how the data is stored. The sections below explores both types of queries.

Basic SELECT statement with LIMIT

A limited select query follows this basic syntax:

```
SELECT fieldA as fa, fieldB as fb, fieldC as fc FROM tableA WHERE fieldC = 'term1
term2' ORDER BY fa desc LIMIT 100
```

We've covered many syntax options with this example, so let's walk through what's possible below.

WHERE Clause and Boolean Predicates

The `WHERE` clause allows Solr's search syntax to be injected into the SQL query. In the example:

```
WHERE fieldC = 'term1 term2'
```

The predicate above will execute a full text search for the phrase 'term1 term2' in fieldC.

To execute a non-phrase query, simply add parens inside of the single quotes. For example:

```
WHERE fieldC = '(term1 term2)'
```

The predicate above searches for `term1` OR `term2` in `fieldC`.

The Solr range query syntax can be used as follows:

```
WHERE fieldC = '[0 TO 100]'
```

Complex boolean queries can be specified as follows:

```
WHERE ((fieldC = 'term1' AND fieldA = 'term2') OR (fieldB = 'term3'))
```

To specify NOT queries, you use the `AND NOT` syntax as follows:

```
WHERE (fieldA = 'term1') AND NOT (fieldB = 'term2')
```

ORDER BY Clause

The `ORDER BY` clause maps directly to Solr fields. Multiple `ORDER BY` fields and directions are supported.

The `score` field is accepted in the `ORDER BY` clause in queries where a limit is specified.

Order by fields are case sensitive.

LIMIT Clause

Limits the result set to the specified size. In the example above the clause `LIMIT 100` will limit the result set to 100 records.

There are a few differences to note between limited and unlimited queries:

- Limited queries support `score` in the field list and `ORDER BY`. Unlimited queries do not.
- Limited queries allow any stored field in the field list. Unlimited queries require the fields to be stored as a DocValues field.
- Limited queries allow any indexed field in the `ORDER BY` list. Unlimited queries require the fields to be stored as a DocValues field.

SELECT DISTINCT Queries

The SQL interface supports both MapReduce and Facet implementations for `SELECT DISTINCT` queries.

The MapReduce implementation shuffles tuples to worker nodes where the Distinct operation is performed. This implementation can perform the Distinct operation over extremely high cardinality fields.

The Facet implementation pushes down the Distinct operation into the search engine using the JSON Facet API. This implementation is designed for high performance, high QPS scenarios on low-to-moderate cardinality fields.

The `aggregationMode` parameter is available in the both the JDBC driver and HTTP interface to choose the underlying implementation (`map_reduce` or `facet`). The SQL syntax is identical for both implementations:

```
SELECT distinct fieldA as fa, fieldB as fb FROM tableA ORDER BY fa desc, fb desc
```

Statistics

The SQL interface supports simple statistics calculated on numeric fields. The supported functions are `count(*)`, `min`, `max`, `sum`, and `avg`.

Because these functions never require data to be shuffled, the aggregations are pushed down into the search

engine and are generated by the StatsComponent.

```
SELECT count(fieldA) as count, sum(fieldB) as sum FROM tableA WHERE fieldC = 'Hello'
```

GROUP BY Aggregations

The SQL interface also supports `GROUP BY` aggregate queries.

As with `SELECT DISTINCT` queries, the SQL interface supports both a MapReduce implementation and a Facet implementation. The MapReduce implementation can build aggregations over extremely high cardinality fields. The Facet implementations provides high performance aggregation over fields with moderate levels of cardinality.

Basic `GROUP BY` with Aggregates

Here is a basic example of a GROUP BY query that requests aggregations:

```
SELECT fieldA as fa, fieldB as fb, count(*) as count, sum(fieldC) as sum,
avg(fieldY) as avg FROM tableA WHERE fieldC = 'term1 term2'
GROUP BY fa, fb HAVING sum > 1000 ORDER BY sum asc LIMIT 100
```

Let's break this down into pieces:

The Column Identifiers and Aliases

The Column Identifiers can contain both fields in the Solr index and aggregate functions. The supported aggregate functions are:

- `count(*)`: Counts the number of records over a set of buckets.
- `sum(field)`: Sums a numeric field over over a set of buckets.
- `avg(field)`: Averages a numeric field over a set of buckets.
- `min(field)`: Returns the min value of a numeric field over a set of buckets.
- `max:(field)`: Returns the max value of a numerics over a set of buckets.

The non-function fields in the field list determine the fields to calculate the aggregations over.

Column aliases are supported for both fields and functions and can be referenced in the GROUP BY, HAVING and ORDER BY clauses.

GROUP BY Clause

The `GROUP BY` clause can contain up to 4 fields in the Solr index. These fields should correspond with the non-function fields in the field list.

HAVING Clause

The `HAVING` clause may contain any function listed in the field list. Complex `HAVING` clauses such as this are supported:

```
SELECT fieldA, fieldB, count(*), sum(fieldC), avg(fieldY)
FROM tableA
WHERE fieldC = 'term1 term2'
GROUP BY fieldA, fieldB
HAVING ((sum(fieldC) > 1000) AND (avg(fieldY) <= 10))
ORDER BY sum(fieldC) asc
LIMIT 100
```

ORDER BY Clause

The `ORDER BY` clause contains any field or function in the field list.

If the `ORDER BY` clause contains the exact fields in the `GROUP BY` clause, then there is no-limit placed on the returned results. If the `ORDER BY` clause contains different fields than the `GROUP BY` clause, a limit of 100 is automatically applied. To increase this limit you must specify a value in the `LIMIT` clause.

Best Practices

Separate Collections

It makes sense to create a separate SolrCloud collection just for the `/sql` handler. This collection can be created using SolrCloud's standard collection API. Since this collection only exists to handle `/sql` requests and provide a pool of worker nodes, this collection does not need to hold any data. Worker nodes are selected randomly from the entire pool of available nodes in the `/sql` handler's collection. So to grow this collection dynamically replicas can be added to existing shards. New replicas will automatically be put to work after they've been added.

Parallel SQL Queries

An earlier section describes how the SQL interface translates the SQL statement to a streaming expression. One of the parameters of the request is the `aggregationMode`, which defines if the query should use a MapReduce-like shuffling technique or push the operation down into the search engine.

Parallelized Queries

The Parallel SQL architecture consists of three logical tiers: a **SQL** tier, a **Worker** tier, and a **Data Table** tier. By default the SQL and Worker tiers are collapsed into the same physical SolrCloud collection.

SQL Tier

The SQL tier is where the `/sql` handler resides. The `/sql` handler takes the SQL query and translates it to a parallel query plan. It then selects worker nodes to execute the plan and sends the query plan to each worker node to be run in parallel.

Once the query plan has been executed by the worker nodes, the `/sql` handler then performs the final merge of the tuples returned by the worker nodes.

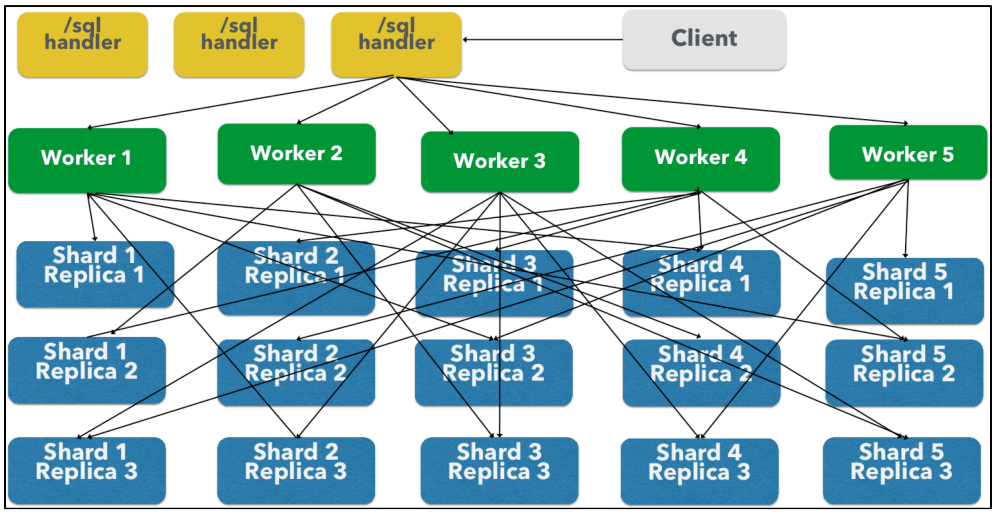
Worker Tier

The workers in the worker tier receive the query plan from the `/sql` handler and execute the parallel query plan. The parallel execution plan includes the queries that need to be made on the Data Table tier and the relational algebra needed to satisfy the query. Each worker node assigned to the query is shuffled 1/N of the tuples from

the Data Tables. The worker nodes execute the query plan and stream tuples back to the worker nodes.

Data Table Tier

The Data Table tier is where the tables reside. Each table is it's own SolrCloud collection. The Data Table layer receives queries from the worker nodes and emits tuples (search results). The Data Table tier also handles the initial sorting and partitioning of tuples sent to the workers. This means the tuples are always sorted and partitioned before they hit the network. The partitioned tuples are sent directly to the correct worker nodes in the proper sort order, ready to be reduced.



The image above shows the three tiers broken out into different SolrCloud collections for clarity. In practice the `/sql` handler and worker collection by default share the same collection.

Note: The image shows the network flow for a single Parallel SQL Query (SQL over MapReduce). This network flow is used when `map_reduce` aggregation mode is used for `GROUP BY` aggregations or the `SELECT DISTINCT` query. The traditional SolrCloud network flow (without workers) is used when the `facet` aggregation mode is used.

Below is a description of the flow:

1. The client sends a SQL query to the `/sql` handler. The request is handled by a single `/sql` handler instance.
2. The `/sql` handler parses the SQL query and creates the parallel query plan.
3. The query plan is sent to worker nodes (in green).
4. The worker nodes execute the plan in parallel. The diagram shows each worker node contacting a collection in the Data Table tier (in blue).
5. The collection in the Data Table tier is the table from the SQL query. Notice that the collection has five shards each with 3 replicas.
6. Notice that each worker contacts one replica from each shard. Because there are 5 workers, each worker is returned 1/5 of the search results from each shard. The partitioning is done inside of the Data Table tier so there is no duplication of data across the network.
7. Also notice with this design ALL replicas in the data layer are shuffling (sorting & partitioning) data simultaneously. As the number of shards, replicas and workers grows this design allows for a massive amount of computing power to be applied to a single query.
8. The worker nodes process the tuples returned from the Data Table tier in parallel. The worker nodes perform the relational algebra needed to satisfy the query plan.
9. The worker nodes stream tuples back to the `/sql` handler where the final merge is done, and finally the tuples are streamed back to the client.

SQL Clients and Database Visualization Tools

The SQL interface supports queries sent from SQL clients and database visualization tools such as DbVisualizer and Apache Zeppelin.

Generic

For most Java based clients, the following jars will need to be placed on the client classpath:

- all .jars found in `$SOLR_HOME/dist/solrj-libs`
- the SolrJ .jar found at `$SOLR_HOME/dist/solr-solrj-<version>.jar`

If you are using Maven, the `org.apache.solr.solr-solrj` artifact contains the required jars.

Once the jars are available on the classpath, the Solr JDBC driver name is `org.apache.solr.client.solrj.io.sql.DriverImpl` and a connection can be made with the following connection string format:

```
jdbc:solr://SOLR_ZK_CONNECTION_STRING?collection=COLLECTION_NAME
```

There are other parameters that can be optionally added to the connection string like `aggregationMode` and `numWorkers`. An example of a Java connection is available in the section JDBC Driver.

DbVisualizer

A step-by-step guide for setting up DbVisualizer is in the section Solr JDBC - DbVisualizer.

SQuirreL SQL

A step-by-step guide for setting up SQuirreL SQL is in the section Solr JDBC - SQuirreL SQL.

Apache Zeppelin (incubating)

A step-by-step guide for setting up Apache Zeppelin is in the section Solr JDBC - Apache Zeppelin.

Solr JDBC - DbVisualizer

- Setup Driver
  - Open Driver Manager
  - Create a New Driver
  - Name the Driver
  - Add Driver Files to Classpath
  - Review and Close Driver Manager
- Create a Connection
  - Use the Connection Wizard
  - Name the Connection
  - Select the Solr driver
  - Specify the Solr URL
- Open and Connect to Solr
- Open SQL Commander to Enter Queries

For DbVisualizer, you will need to create a new driver for Solr using the DbVisualizer Driver Manager. This will add several SolrJ client .jars to the DbVisualizer classpath. The files required are:

- all .jars found in `$SOLR_HOME/dist/solrj-lib`

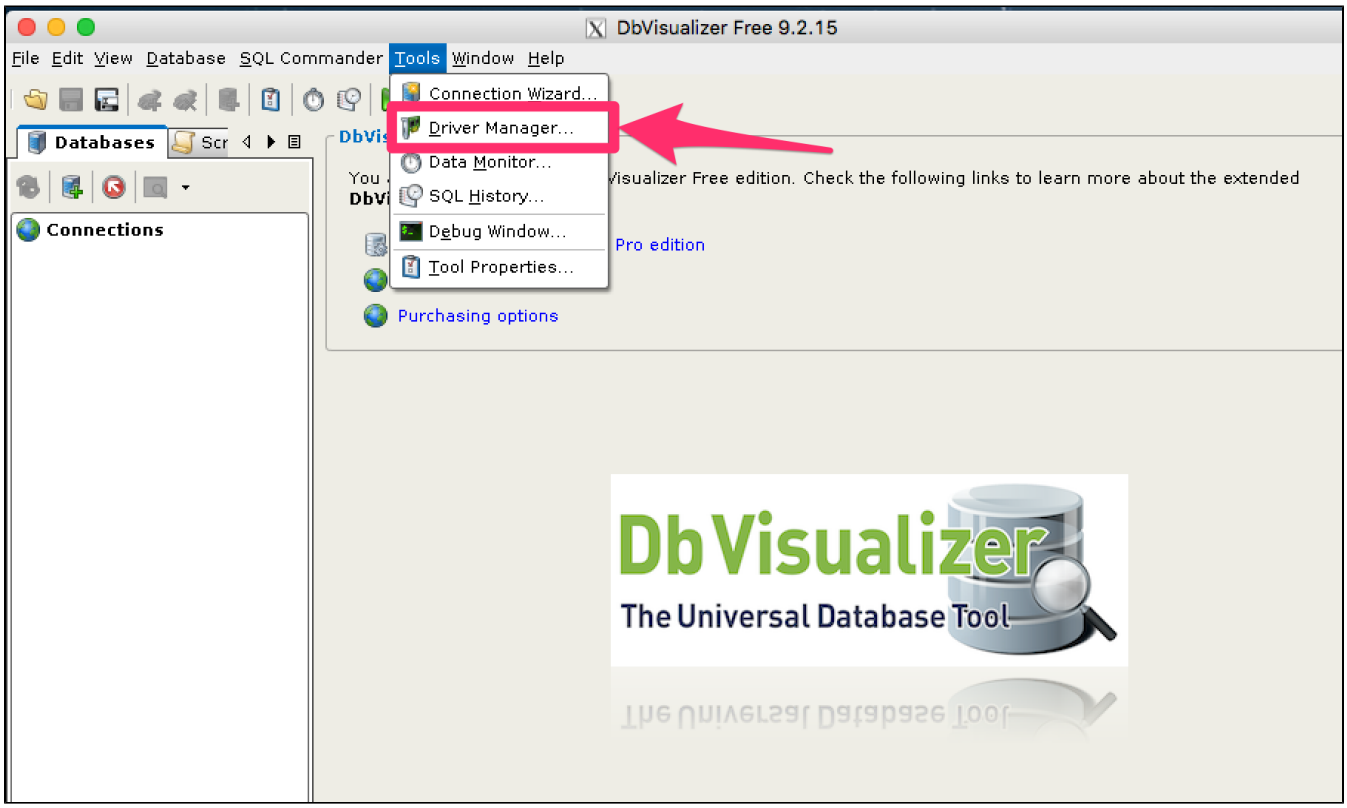
- the SolrJ .jar found at `$SOLR_HOME/dist/solr-solrj-<version>.jar`

Once the driver has been created, you can create a connection to Solr with the connection string format outlined in the generic section and use the SQL Commander to issue queries.

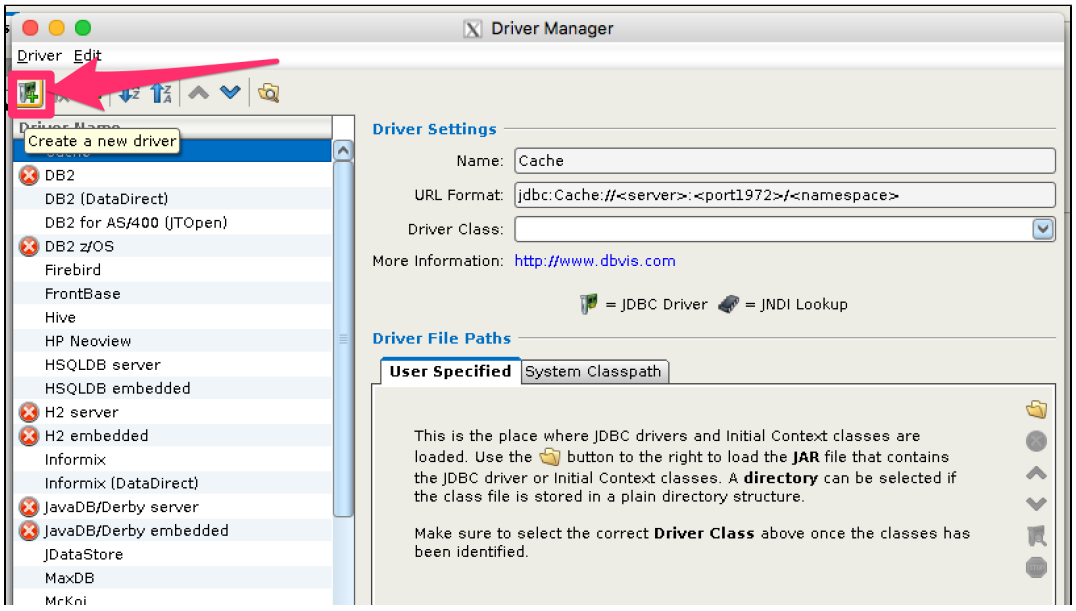
Setup Driver

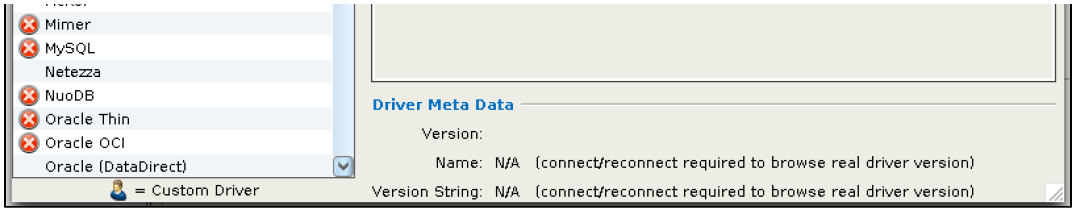
Open Driver Manager

From the Tools menu, choose Driver Manager to add a driver.



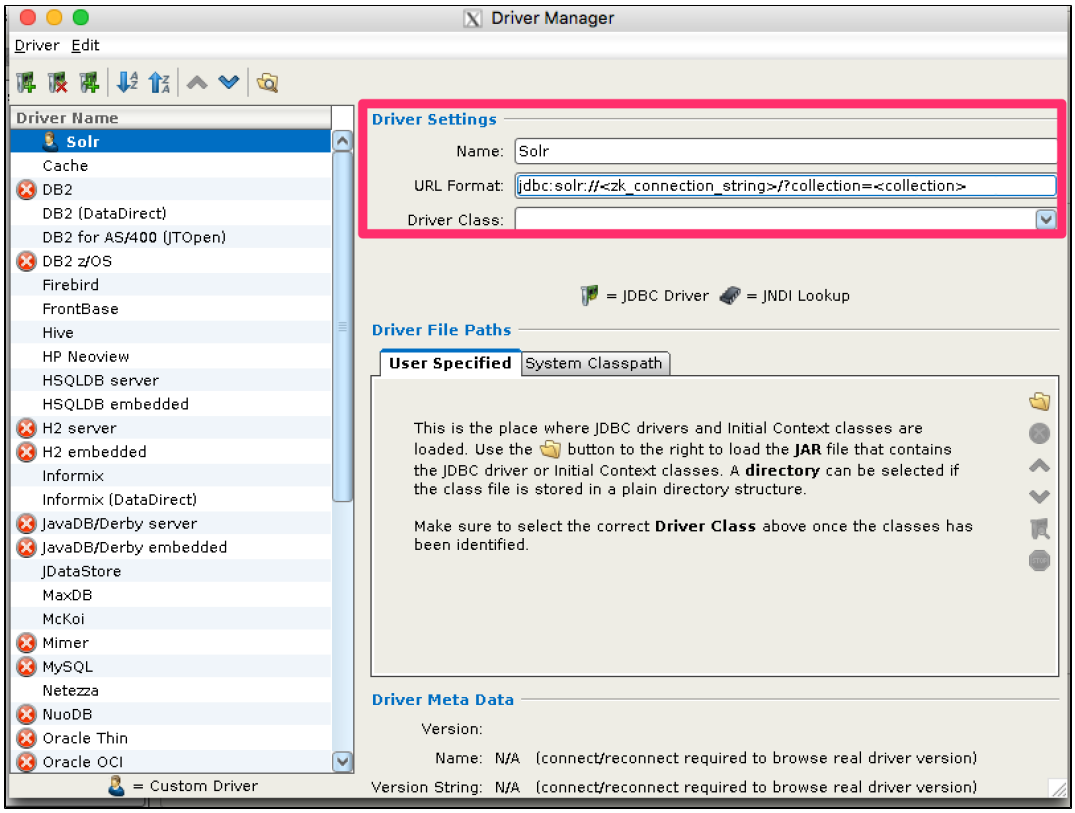
Create a New Driver





Name the Driver

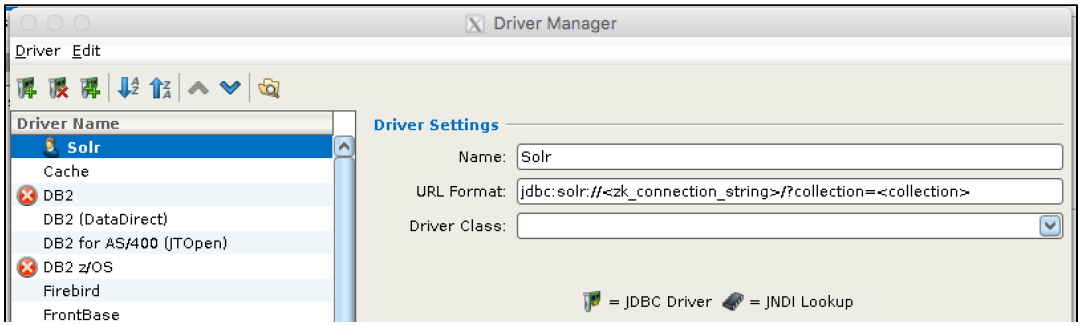
Provide a name for the driver, and provide the URL format: `jdbc:solr://<zk_connection_string>/?collection=<collection>`. Do not fill in values for the variables "`zk_connection_string`" and "`collection`", those will be provided later when the connection to Solr is configured. The Driver Class will also be automatically added when the driver .jars are added.

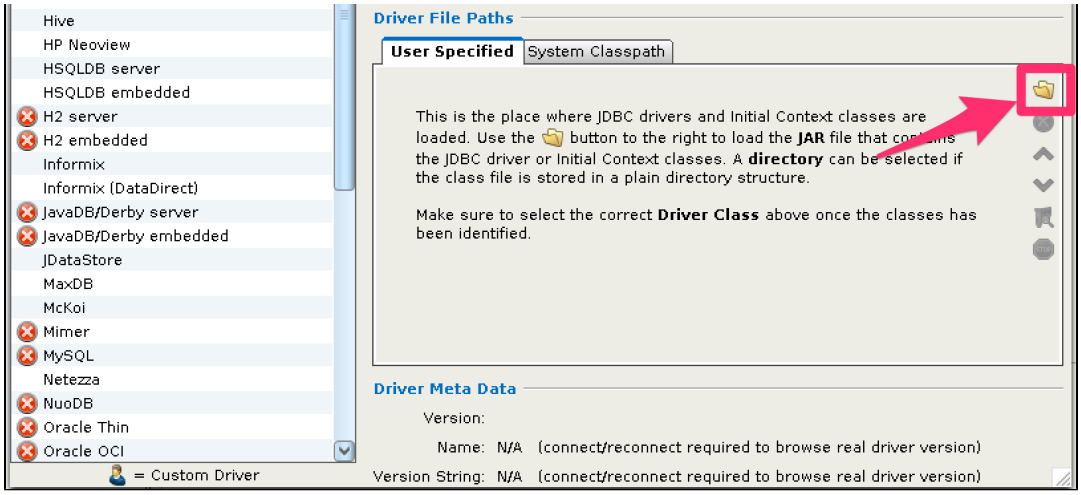


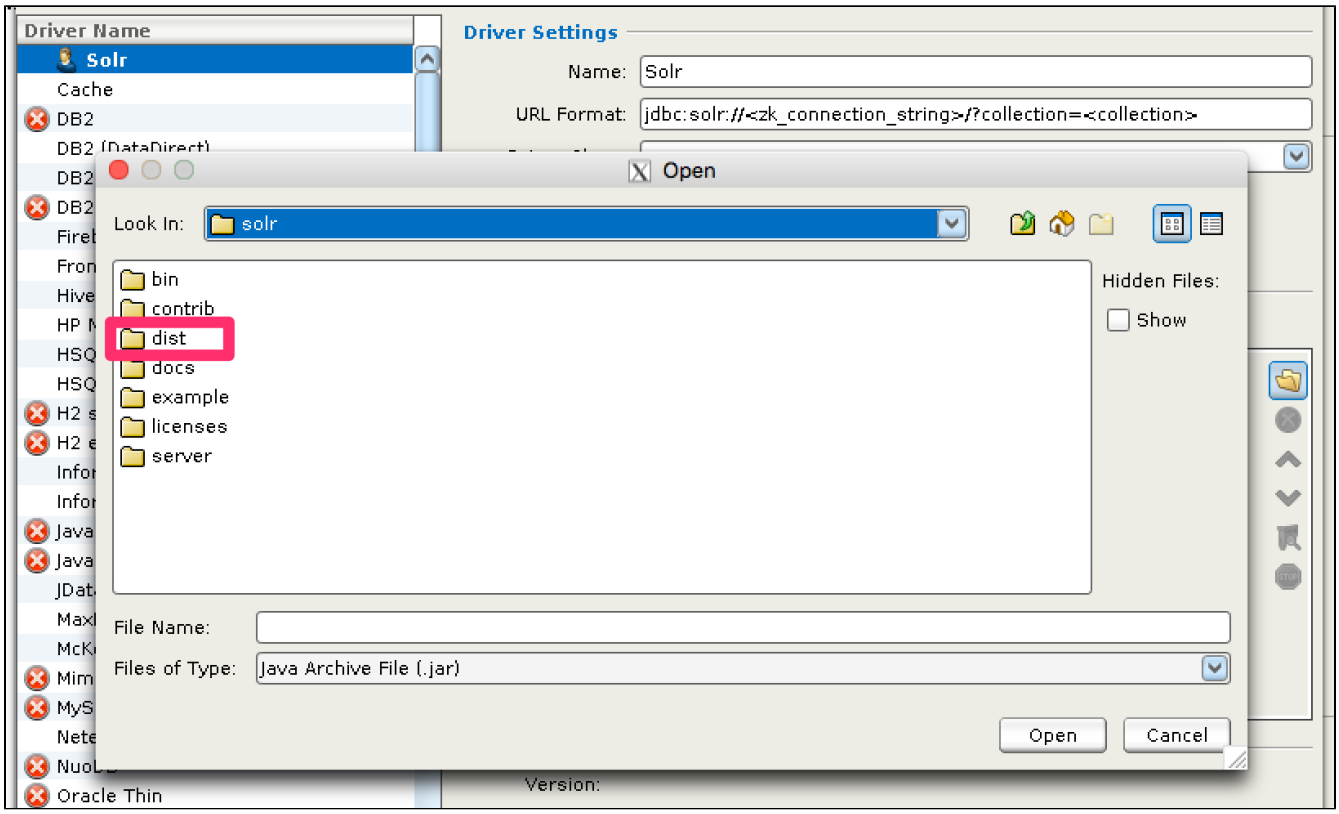
Add Driver Files to Classpath

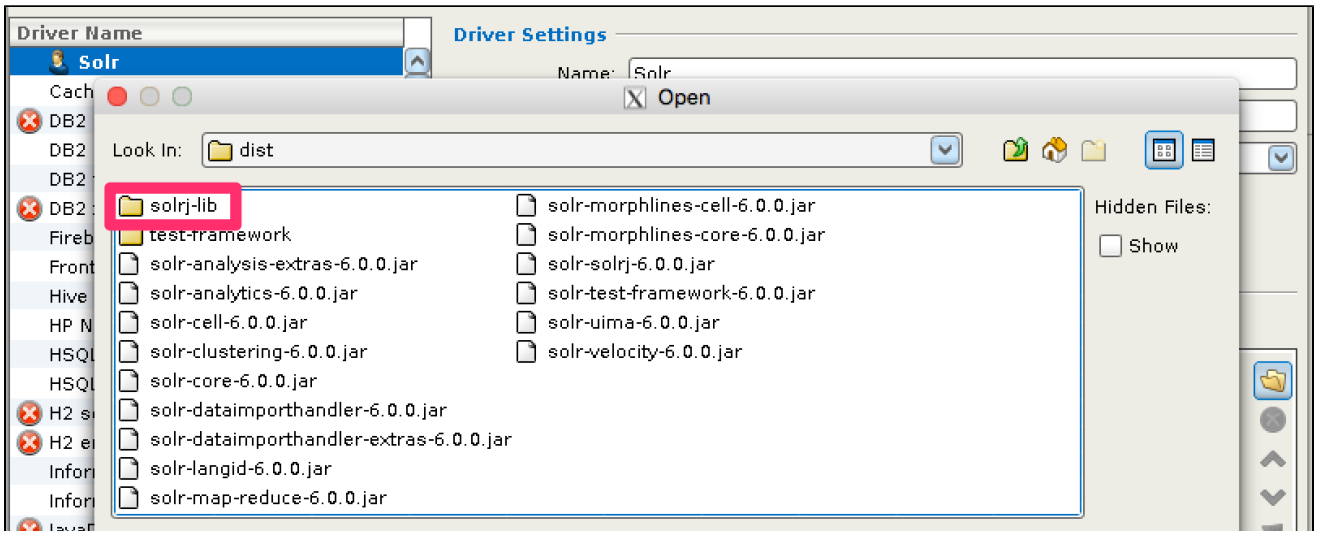
The driver files to be added are:

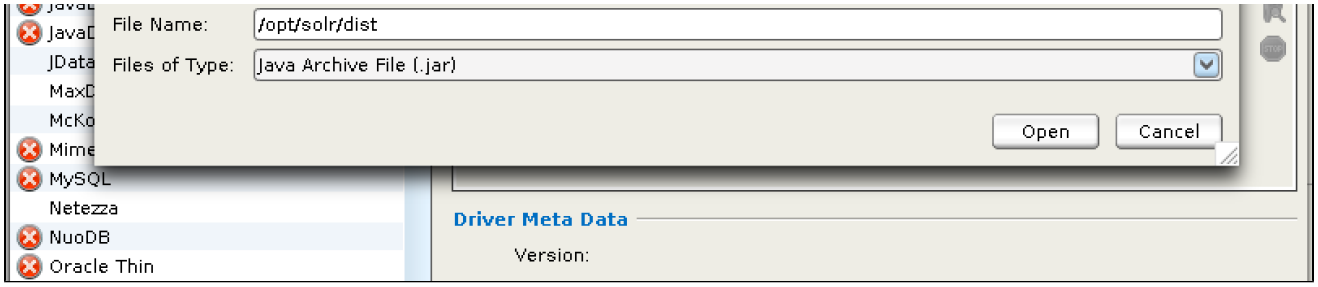
- all .jars in `$SOLR_HOME/dist/solrj-lib`
- the SolrJ .jar found in `$SOLR_HOME/dist/solr-solrj-<version>.jar`

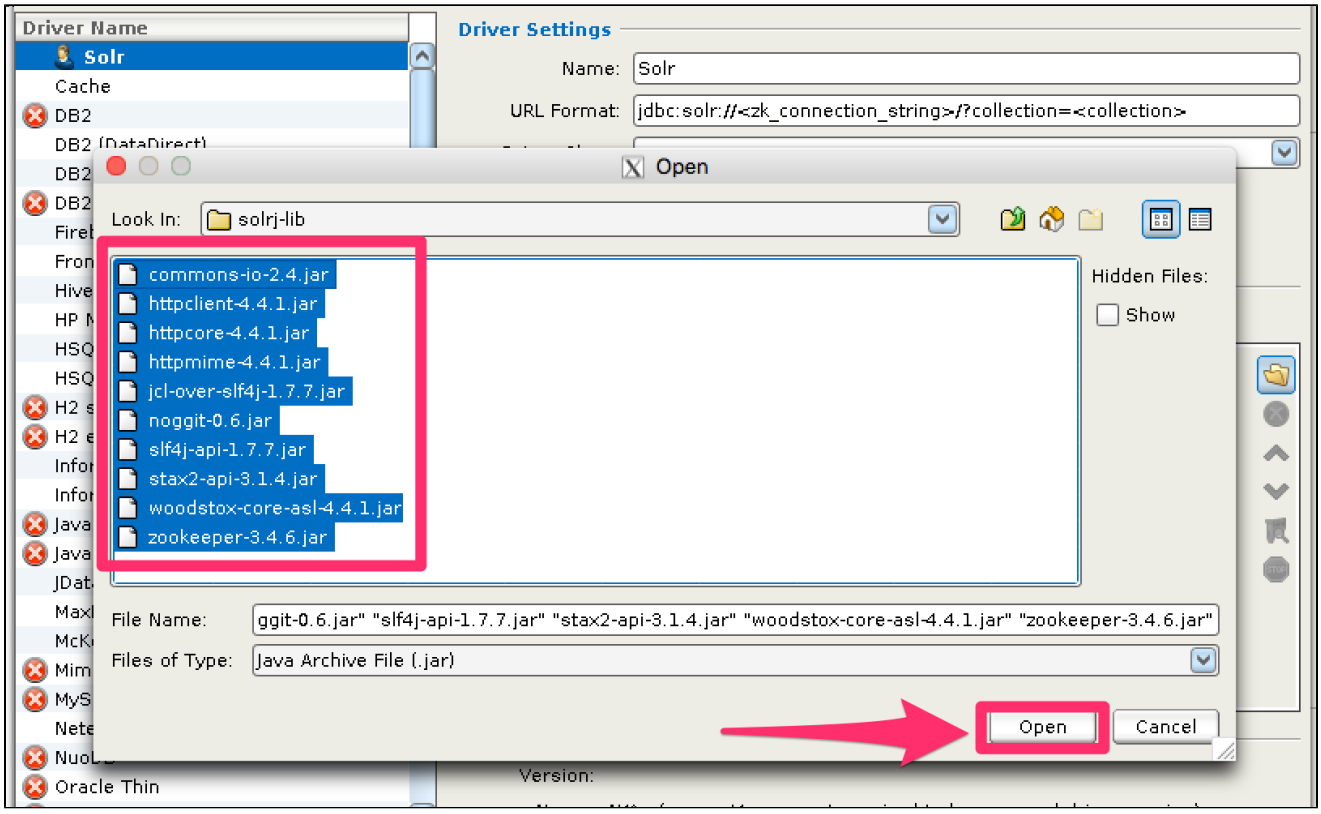


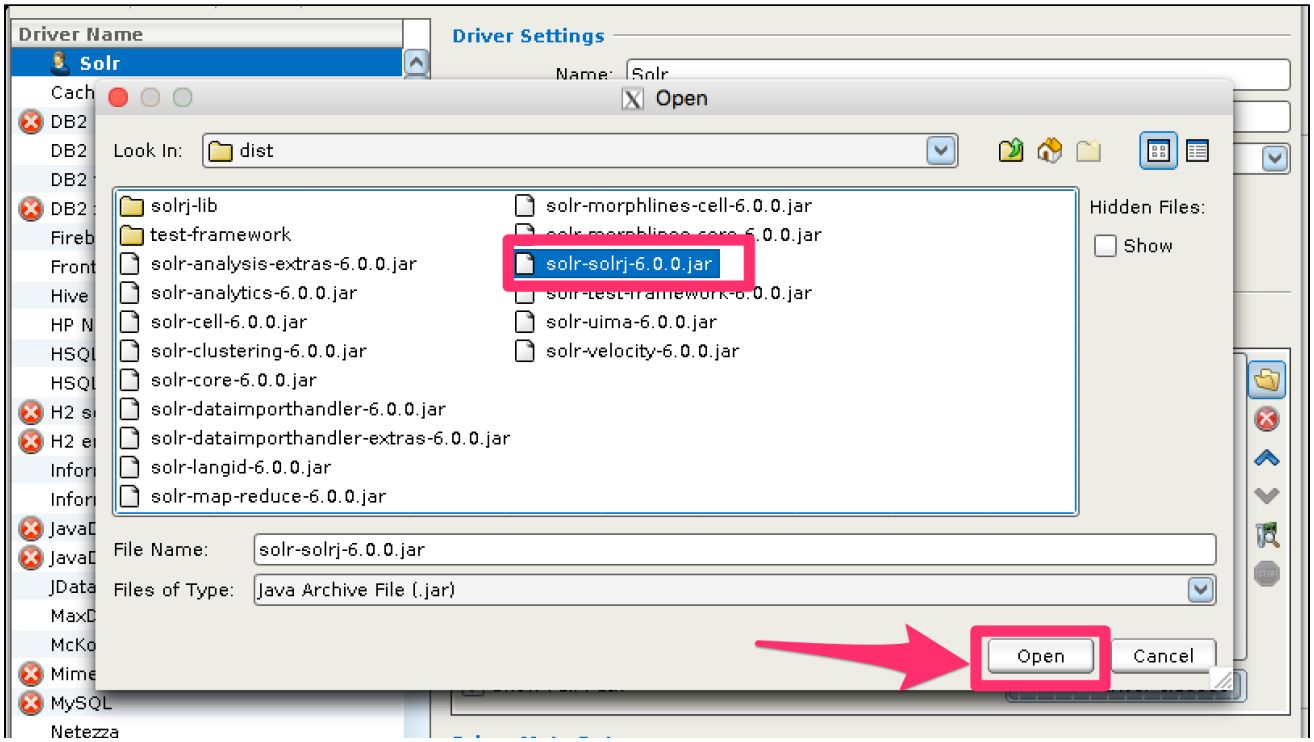

Hive
HP Neoview
HSQLDB server
HSQLDB embedded
H2 server
H2 embedded
Informix
Informix (DataDirect)
JavaDB/Derby server
JavaDB/Derby embedded
JDataStore
MaxDB
McKoi
Mimer
MySQL
Netezza
NuoDB
Oracle Thin
Oracle OCI
= Custom Driver
Driver File Paths
User Specified
System Classpath
This is the place where JDBC drivers and Initial Context classes are loaded. Use the button to the right to load the JAR file that contains the JDBC driver or Initial Context classes. A directory can be selected if the class file is stored in a plain directory structure.
Make sure to select the correct Driver Class above once the classes has been identified.
Driver Meta Data
Version:
Name: N/A (connect/reconnect required to browse real driver version)
Version String: N/A (connect/reconnect required to browse real driver version)


Driver Name
Solr
Cache
DB2
Oracle Thin
Driver Settings
Name: Solr
URL Format: jdbc:solr://<zk_connection_string>/?collection=<collection>
Version:
Open
Look In: solr
bin
contrib
dist
docs
example
licenses
server
Hidden Files:
Show
File Name:
Files of Type: Java Archive File (.jar)
Open
Cancel


Driver Name
Solr
Driver Settings
Open
Look In: dist
solrj-lib
test-framework
solr-analysis-extras-6.0.0.jar
solr-analytics-6.0.0.jar
solr-cell-6.0.0.jar
solr-clustering-6.0.0.jar
solr-core-6.0.0.jar
solr-dataimporthandler-6.0.0.jar
solr-dataimporthandler-extras-6.0.0.jar
solr-langid-6.0.0.jar
solr-map-reduce-6.0.0.jar
solr-morphlines-cell-6.0.0.jar
solr-morphlines-core-6.0.0.jar
solr-solrj-6.0.0.jar
solr-test-framework-6.0.0.jar
solr-uima-6.0.0.jar
solr-velocity-6.0.0.jar
Hidden Files:
Show


File Name:
/opt/solr/dist
Files of Type:
Java Archive File (.jar)
Open
Cancel
Driver Meta Data
Version:


Driver Name
Solr
Driver Settings
Name:
Solr
URL Format:
jdbc:solr://<zk_connection_string>/?collection=<collection>
Open
Look In:
solrj-lib
commons-io-2.4.jar
httpclient-4.4.1.jar
httpcore-4.4.1.jar
httpmime-4.4.1.jar
jcl-over-slf4j-1.7.7.jar
noggit-0.6.jar
slf4j-api-1.7.7.jar
stax2-api-3.1.4.jar
woodstox-core-asl-4.4.1.jar
zookeeper-3.4.6.jar
Hidden Files:
Show
File Name:
ggit-0.6.jar" "slf4j-api-1.7.7.jar" "stax2-api-3.1.4.jar" "woodstox-core-asl-4.4.1.jar" "zookeeper-3.4.6.jar"
Files of Type:
Java Archive File (.jar)
Open
Cancel
Version:


Driver Name
Solr
Driver Settings
Open
Look In:
dist
solrj-lib
test-framework
solr-analysis-extras-6.0.0.jar
solr-analytics-6.0.0.jar
solr-cell-6.0.0.jar
solr-clustering-6.0.0.jar
solr-core-6.0.0.jar
solr-dataimporthandler-6.0.0.jar
solr-dataimporthandler-extras-6.0.0.jar
solr-langid-6.0.0.jar
solr-map-reduce-6.0.0.jar
solr-morphlines-cell-6.0.0.jar
solr-solrj-6.0.0.jar
solr-uima-6.0.0.jar
solr-velocity-6.0.0.jar
Hidden Files:
Show
File Name:
solr-solrj-6.0.0.jar
Files of Type:
Java Archive File (.jar)
Open
Cancel



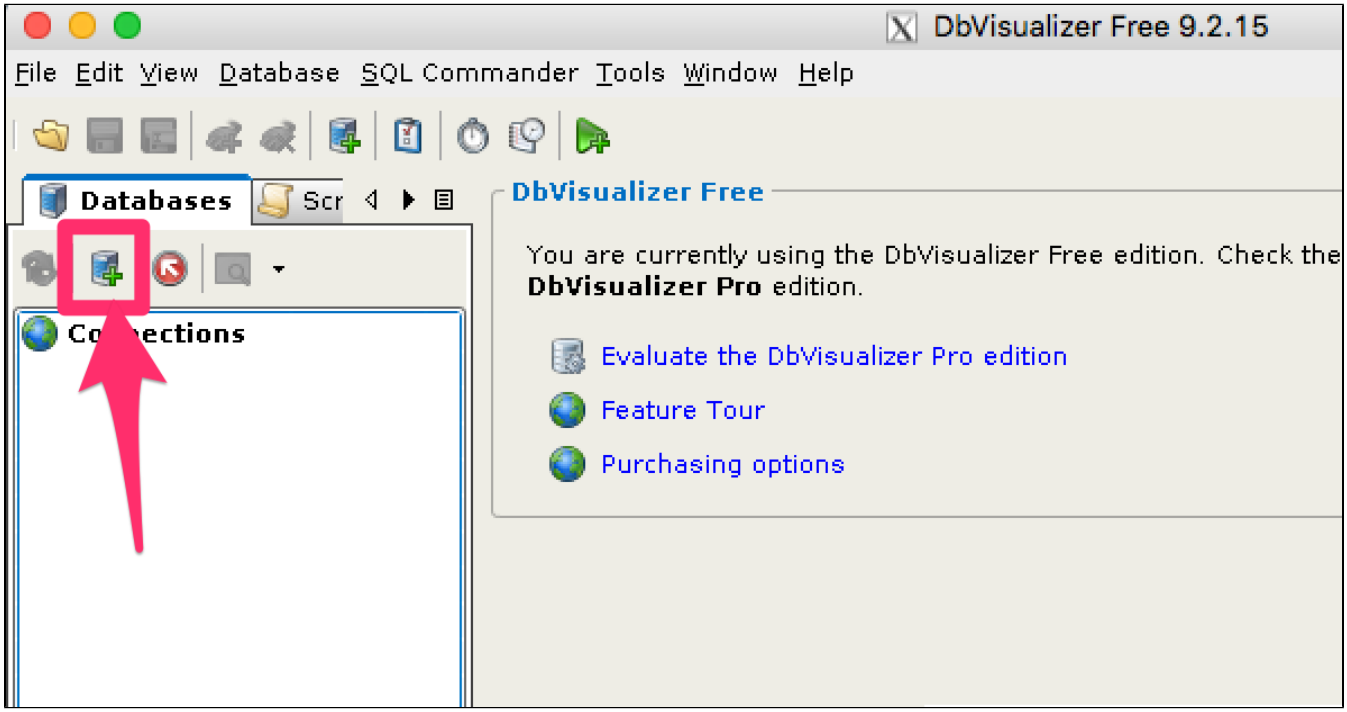
Review and Close Driver Manager

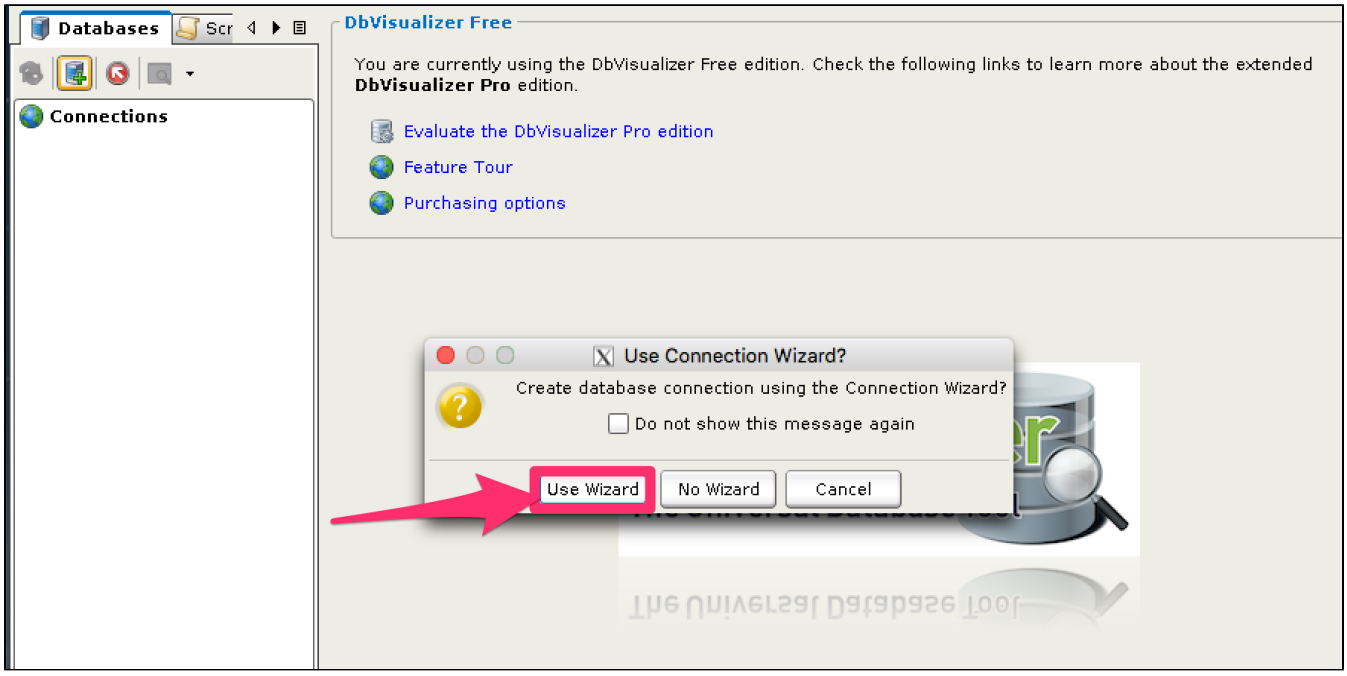
Once the driver files have been added, you can close the Driver Manager.

Create a Connection

Next, create a connection to Solr using the driver just created.

Use the Connection Wizard

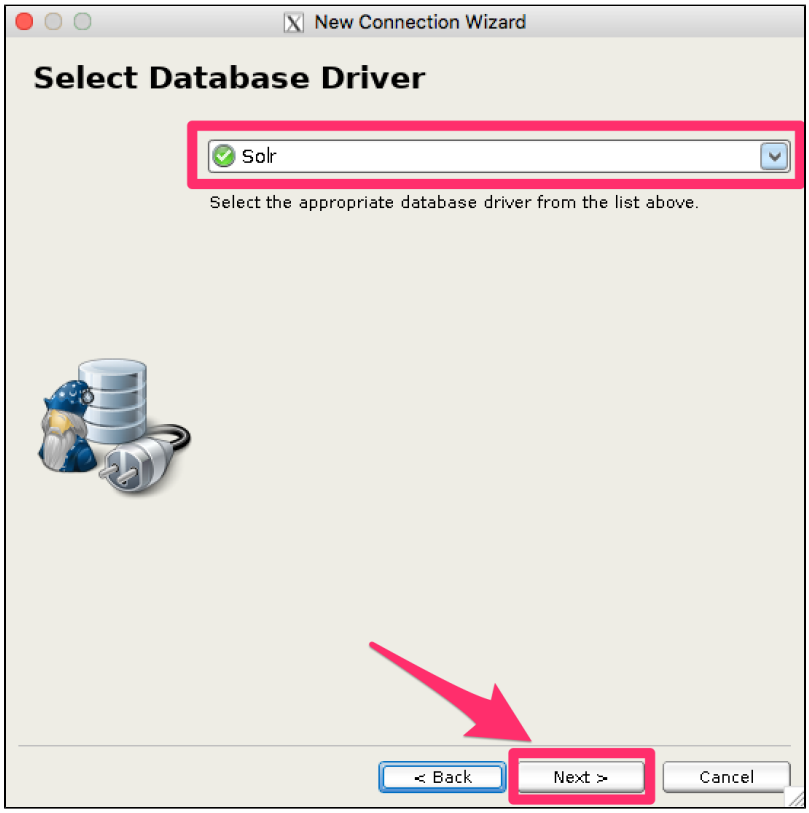




Name the Connection

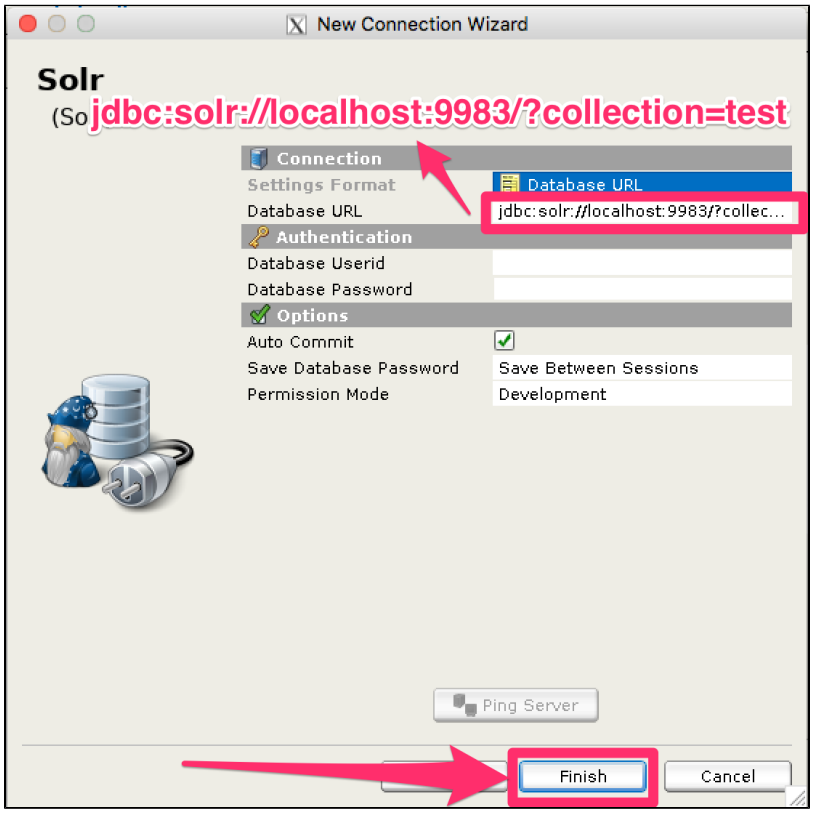


Select the Solr driver



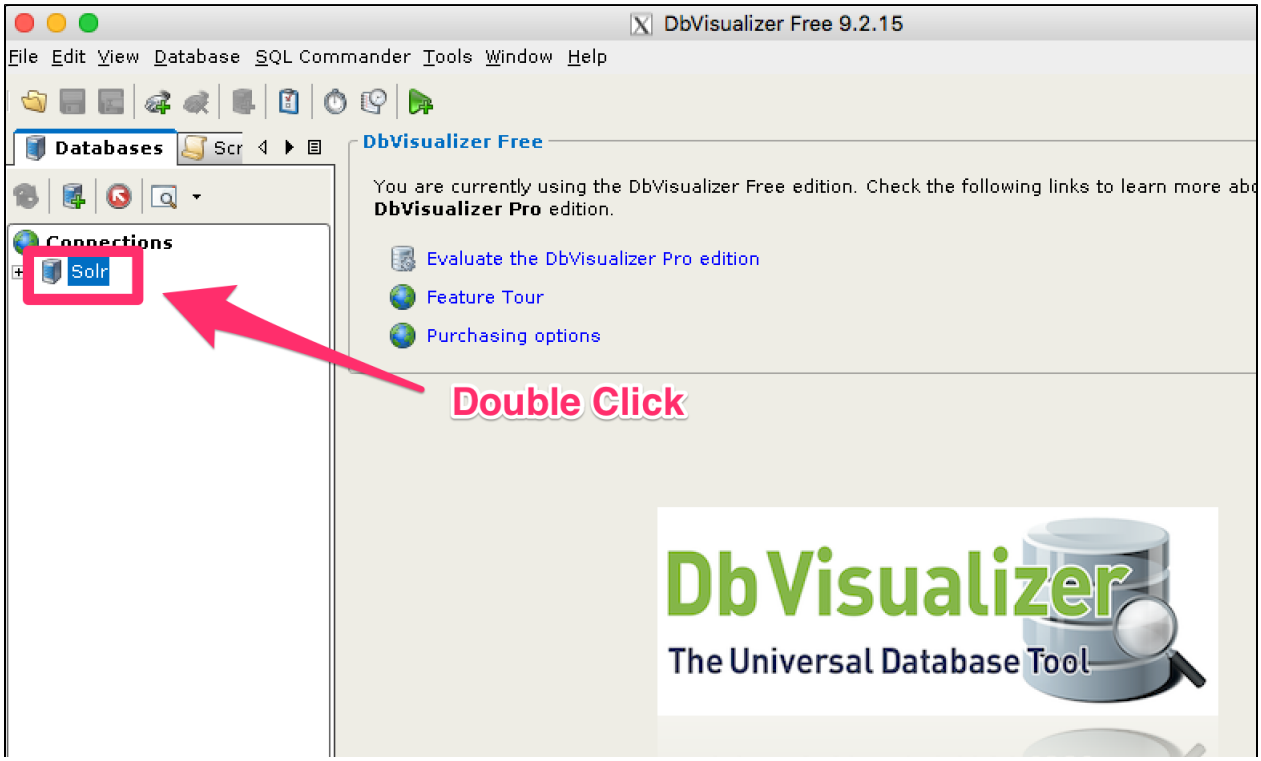
Specify the Solr URL

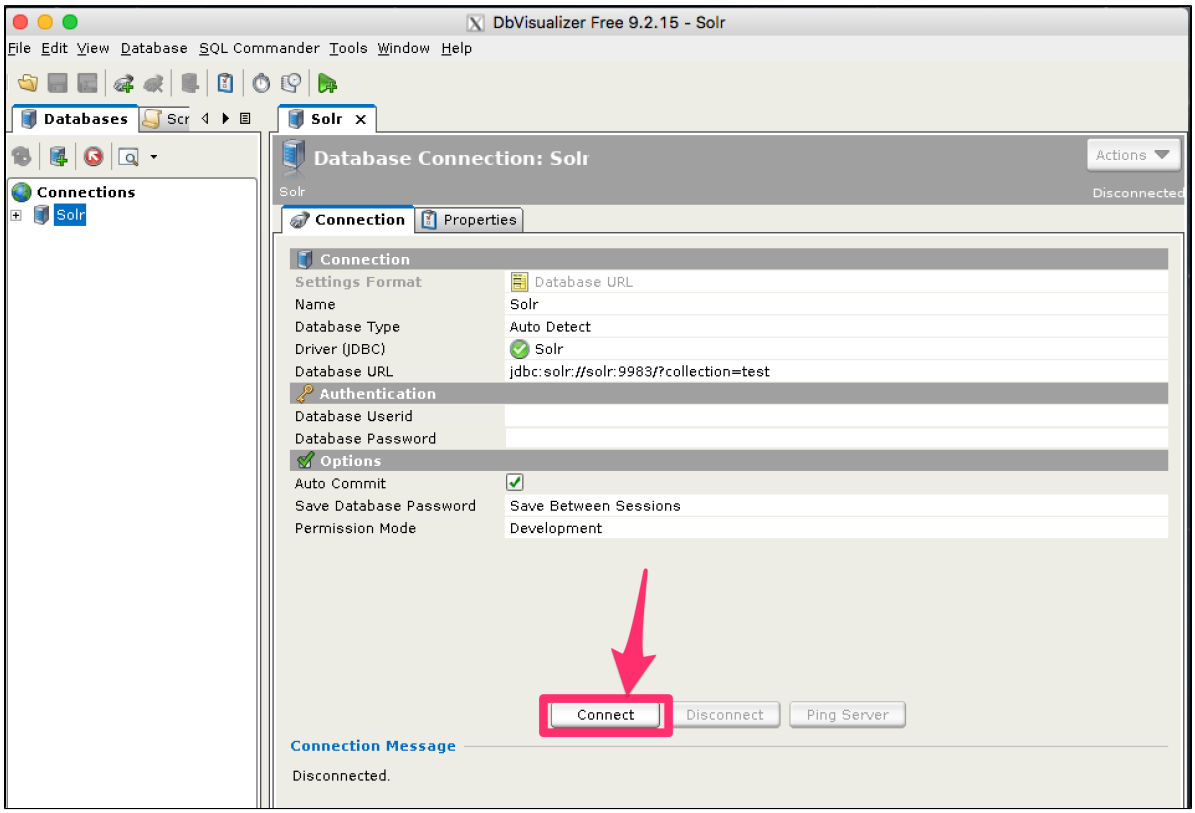
Provide the Solr URL, using the ZooKeeper host and port and the collection. For example, `jdbc:solr://localhost:9983?collection=test`



Open and Connect to Solr

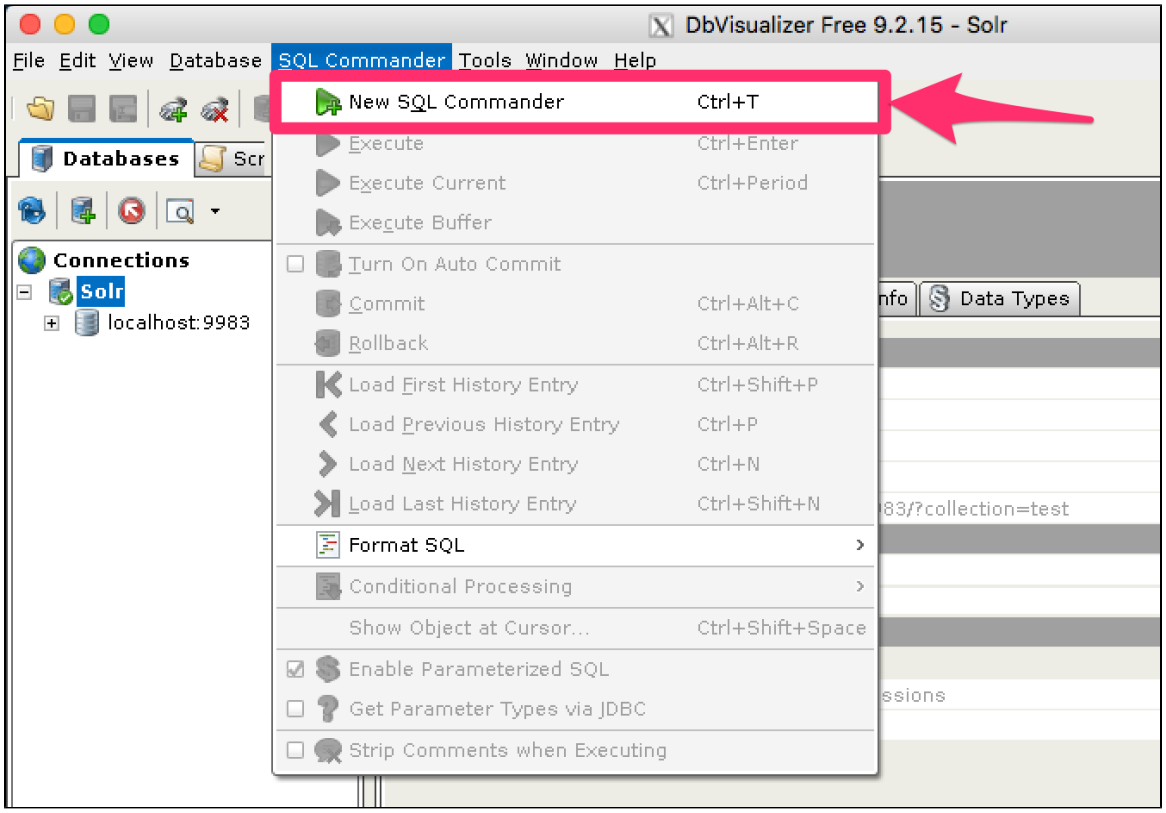
Once the connection has been created, double-click on it to open the connection details screen and connect to Solr.

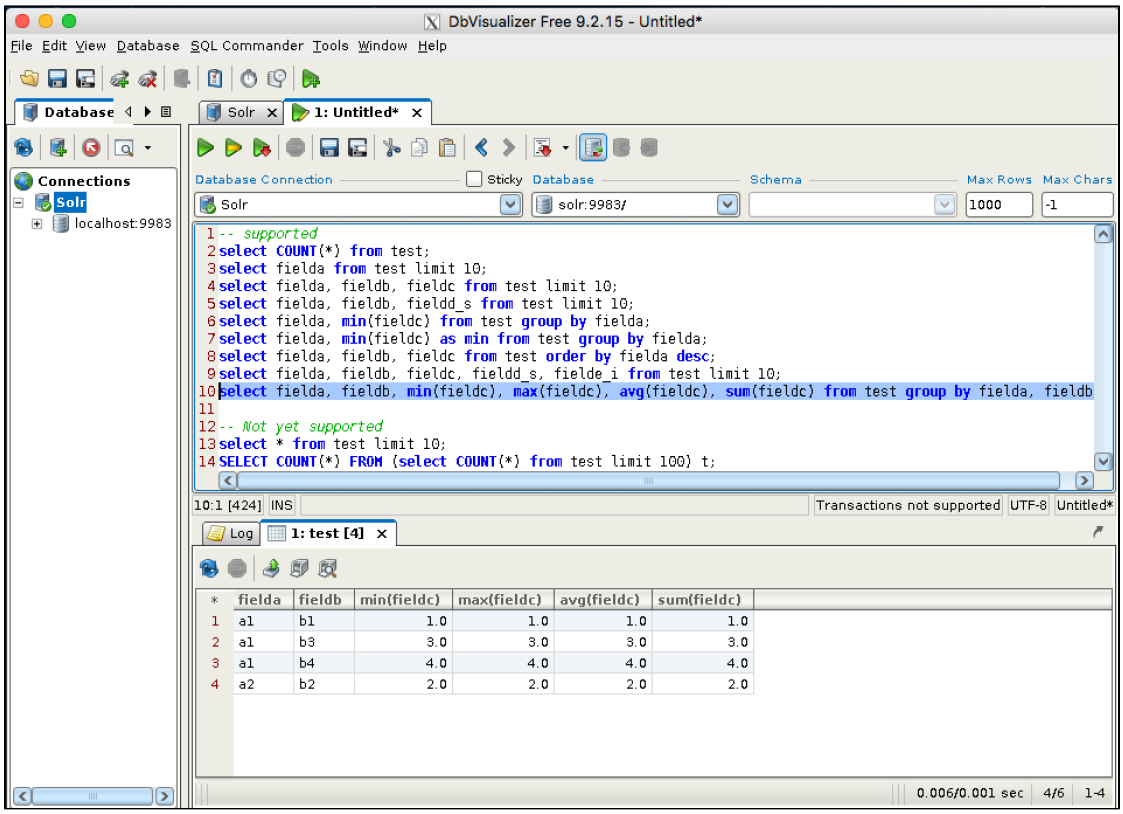




Open SQL Commander to Enter Queries

When the connection is established, you can use the SQL Commander to issue queries and view data.





Solr JDBC - SQuirreL SQL

- Add Solr JDBC Driver
  - Open Drivers
  - Add Driver
  - Name the Driver
  - Add Solr JDBC jars to Classpath
  - Add the Solr JDBC driver class name
- Create an Alias
  - Open Aliases
  - Add an Alias
  - Configure the Alias
  - Connect to the Alias
- Querying

For SQuirreL SQL, you will need to create a new driver for Solr. This will add several SolrJ client .jars to the SQuirreL SQL classpath. The files required are:

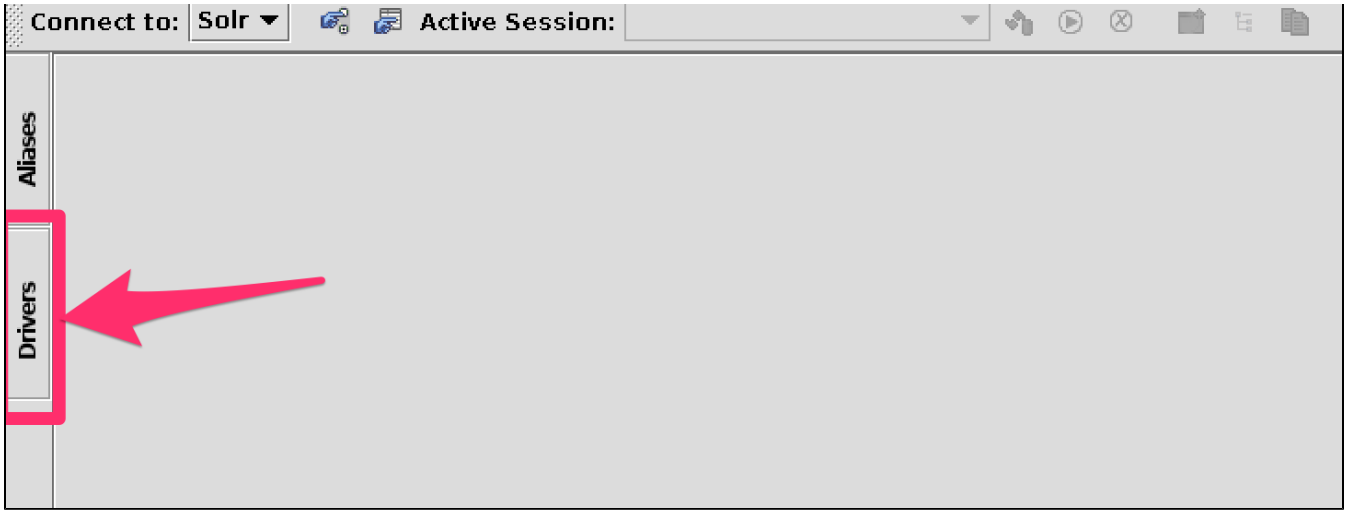
- all .jars found in `$SOLR_HOME/dist/solrj-libs`
- the SolrJ .jar found at `$SOLR_HOME/dist/solr-solrj-<version>.jar`

Once the driver has been created, you can create a connection to Solr with the connection string format outlined in the generic section and use the editor to issue queries.

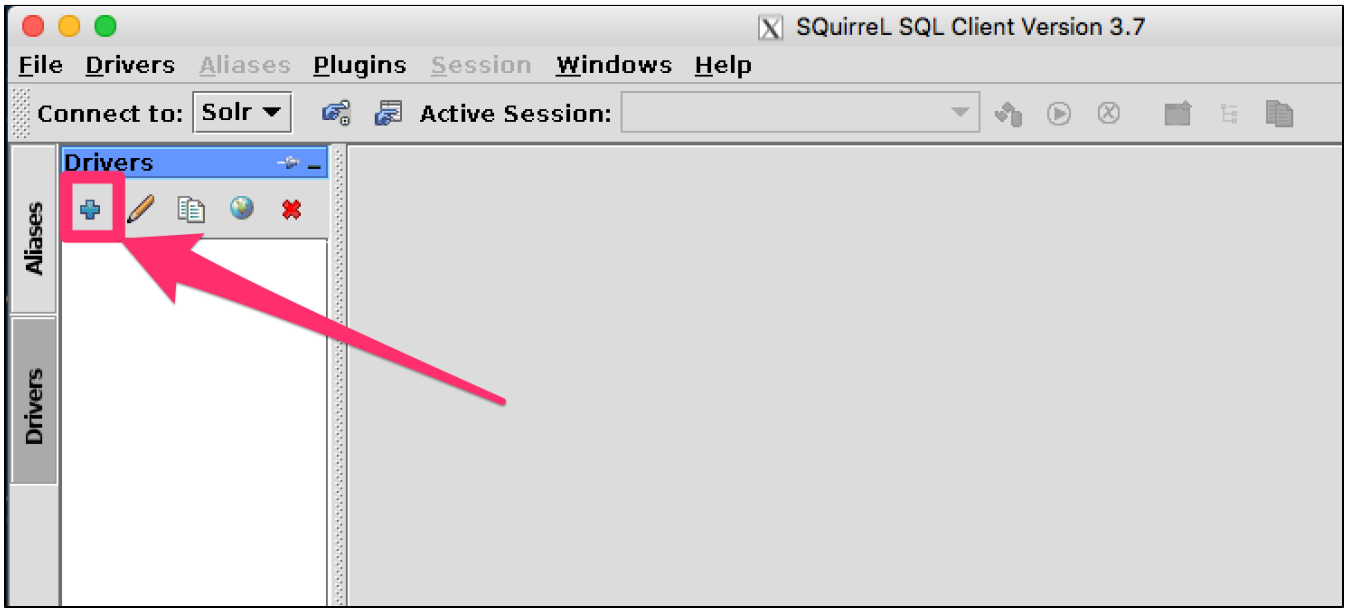
Add Solr JDBC Driver

Open Drivers



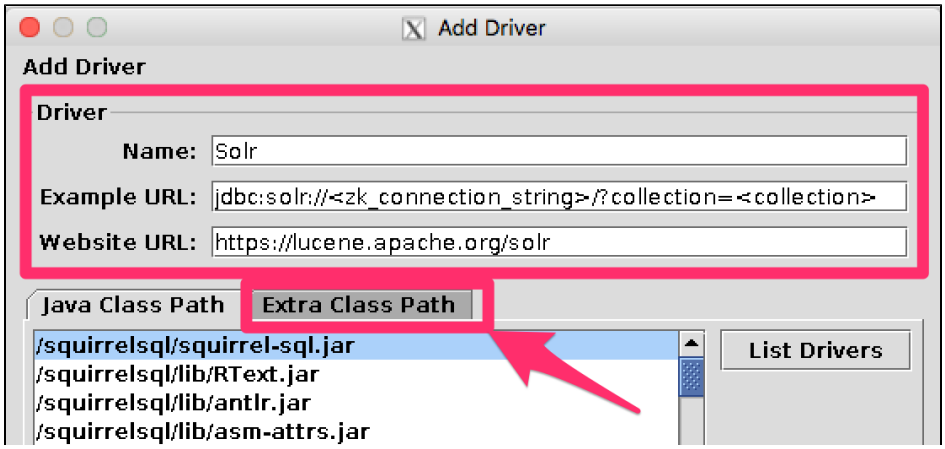


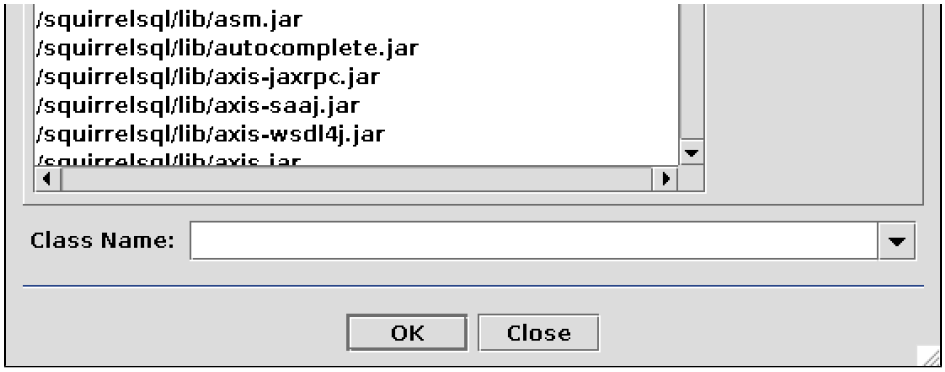
Add Driver



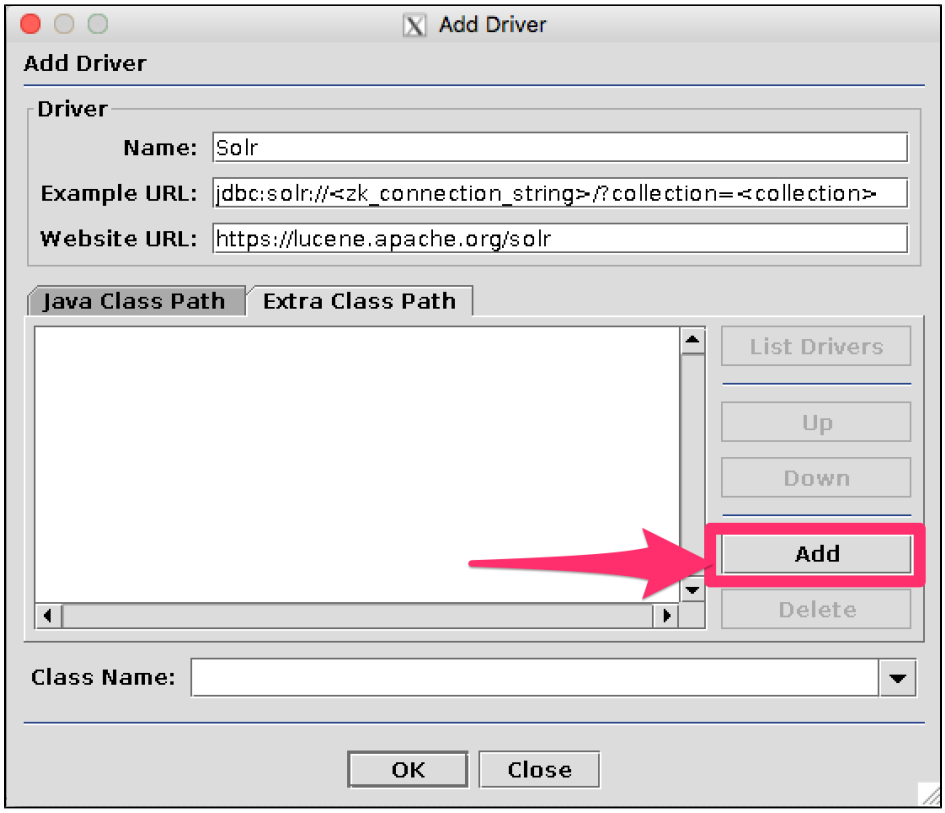
Name the Driver

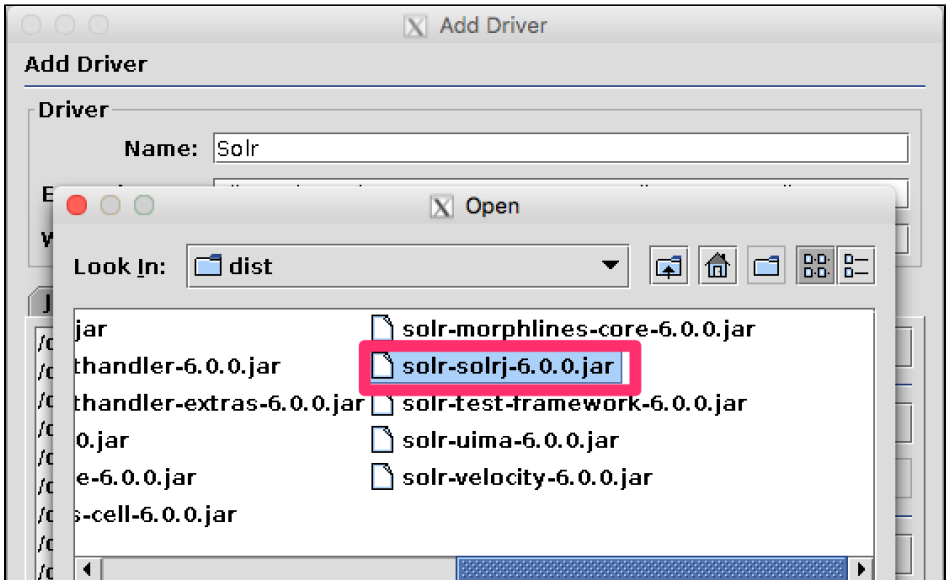
Provide a name for the driver, and provide the URL format: `jdbc:solr://<zk_connection_string>/?collection=<collection>`. Do not fill in values for the variables "`zk_connection_string`" and "`collection`", those will be defined later when the connection to Solr is configured.

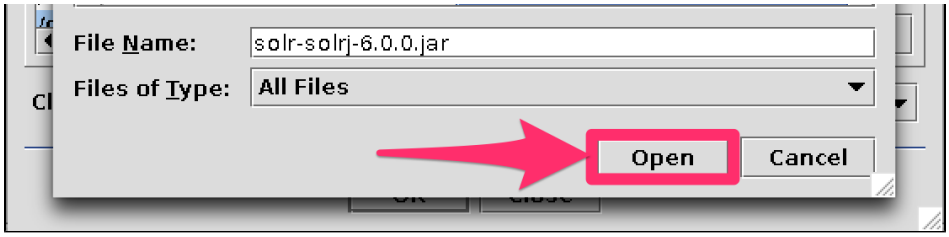


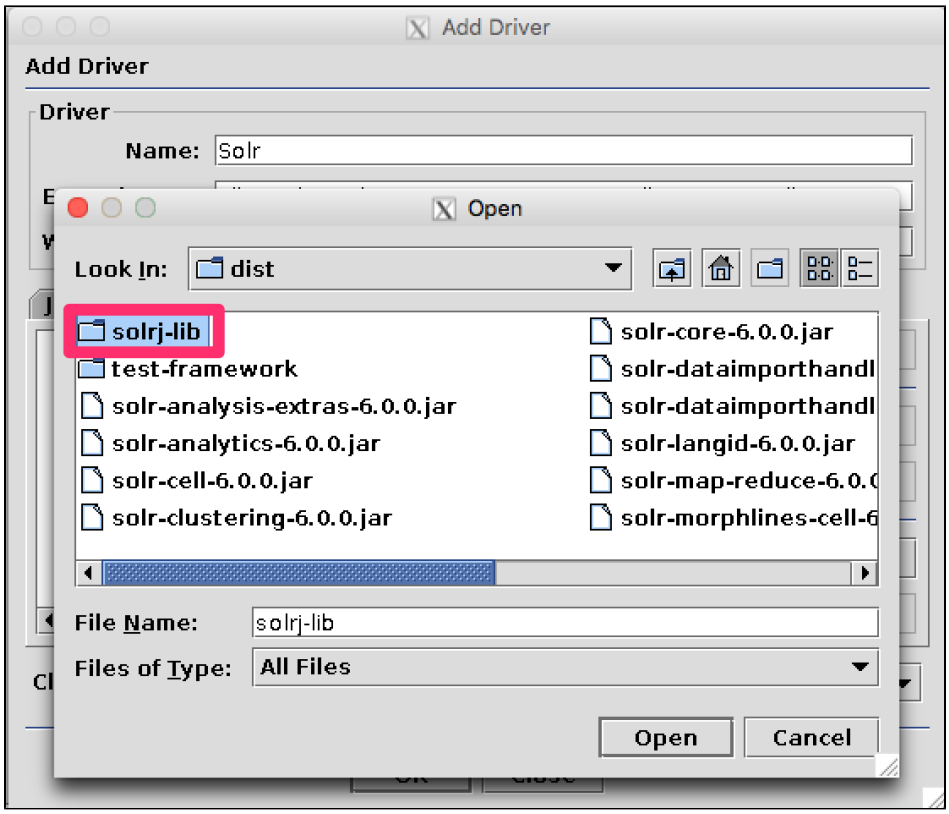


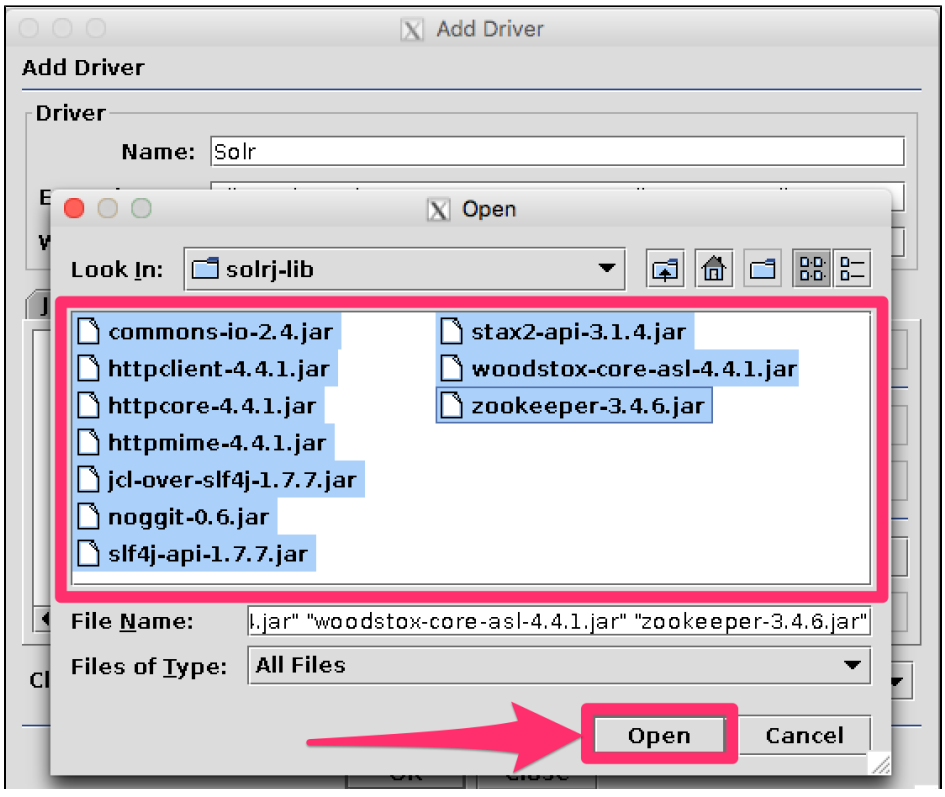
Add Solr JDBC jars to Classpath





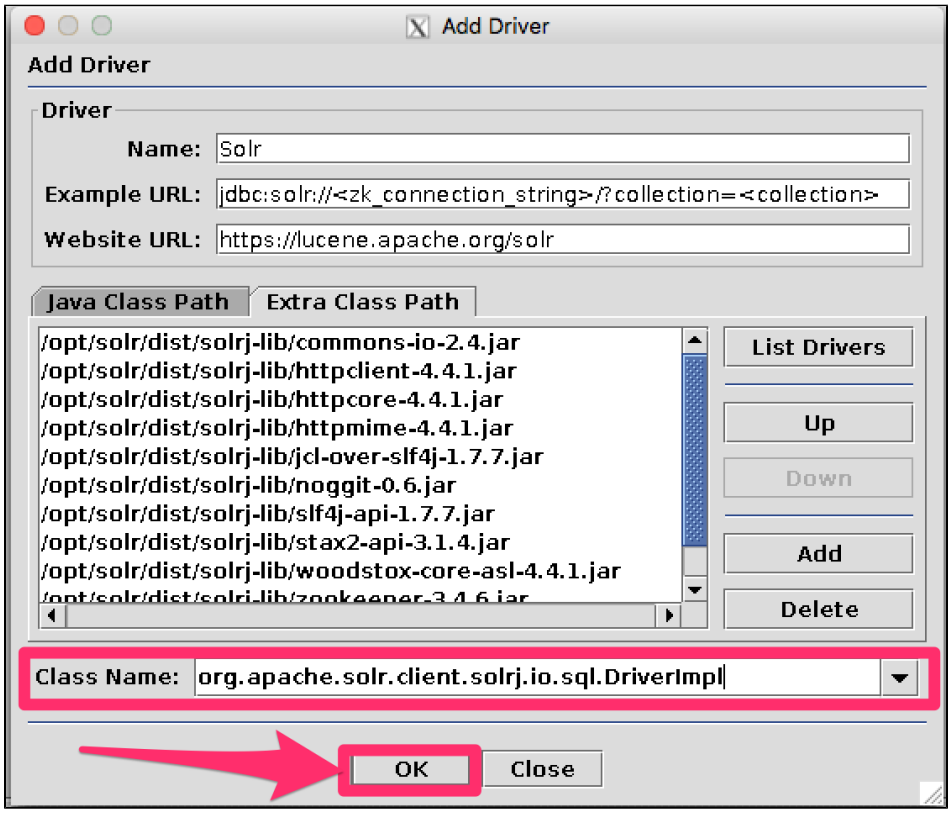

File Name:
solr-solrj-6.0.0.jar
Files of Type:
All Files
Open
Cancel


Add Driver
Add Driver
Driver
Name:
Solr
Open
Look In:
dist
solrj-lib
test-framework
solr-analysis-extras-6.0.0.jar
solr-analytics-6.0.0.jar
solr-cell-6.0.0.jar
solr-clustering-6.0.0.jar
solr-core-6.0.0.jar
solr-dataimporthandl
solr-dataimporthandl
solr-langid-6.0.0.jar
solr-map-reduce-6.0.0
solr-morphlines-cell-6
File Name:
solrj-lib
Files of Type:
All Files
Open
Cancel


Add Driver
Add Driver
Driver
Name:
Solr
Open
Look In:
solrj-lib
commons-io-2.4.jar
httpclient-4.4.1.jar
httpcore-4.4.1.jar
httpmime-4.4.1.jar
jcl-over-slf4j-1.7.7.jar
noggit-0.6.jar
slf4j-api-1.7.7.jar
stax2-api-3.1.4.jar
woodstox-core-asl-4.4.1.jar
zookeeper-3.4.6.jar
File Name:
l.jar" "woodstox-core-asl-4.4.1.jar" "zookeeper-3.4.6.jar"
Files of Type:
All Files
Open
Cancel

Add the Solr JDBC driver class name

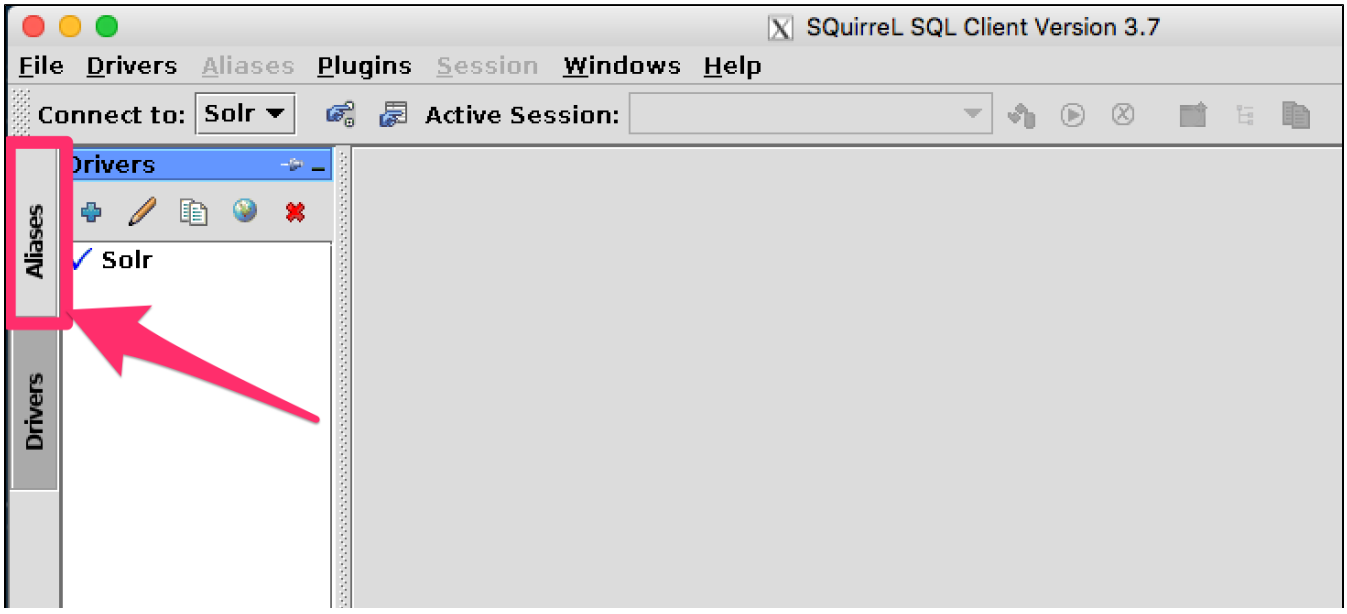
After adding the .jars, you will need to additionally define the Class Name `org.apache.solr.client.solrj.io.sql.DriverImpl`.



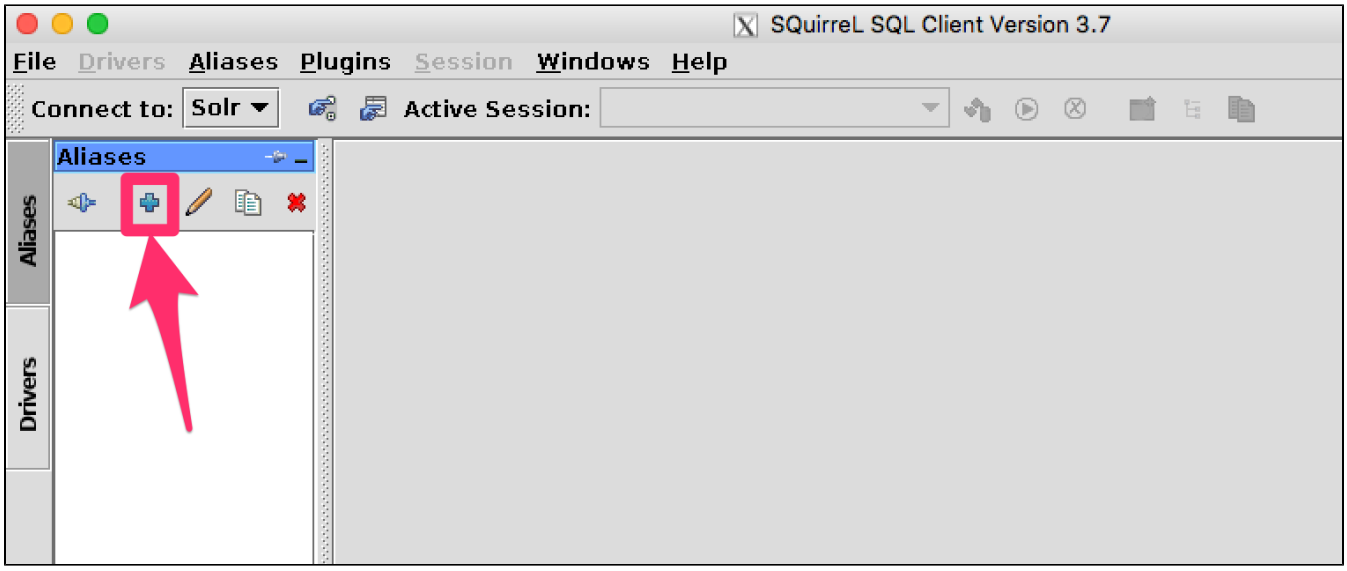
Create an Alias

To define a JDBC connection, you must define an alias.

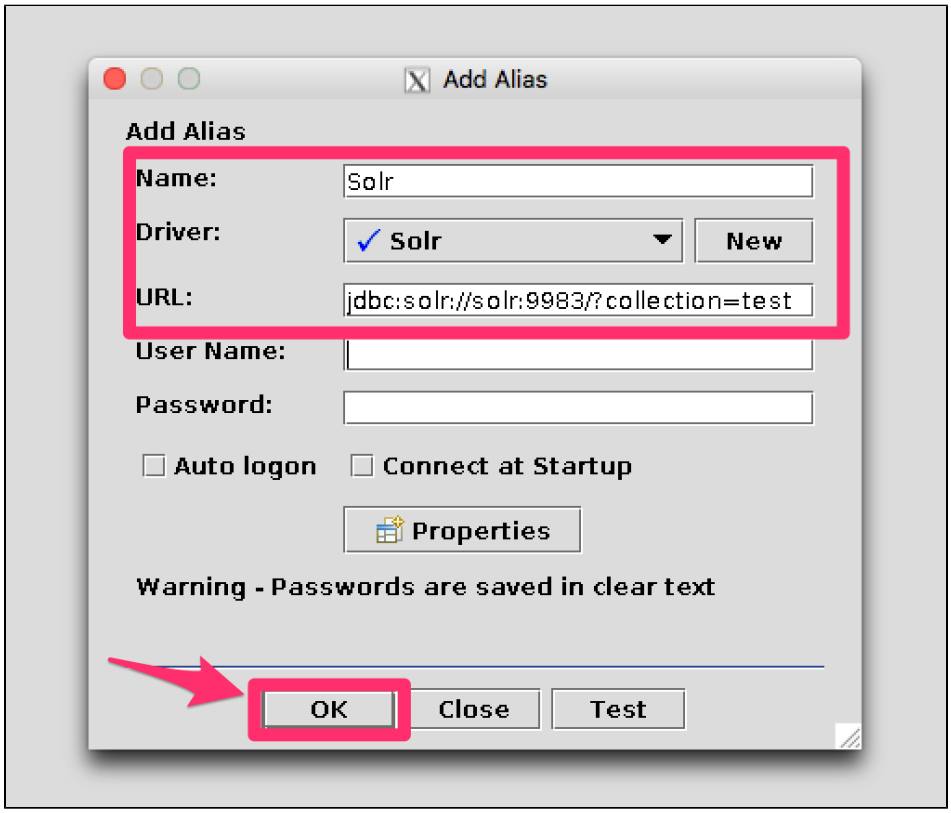
Open Aliases



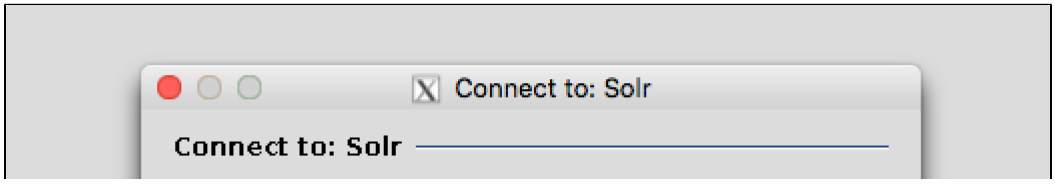
Add an Alias

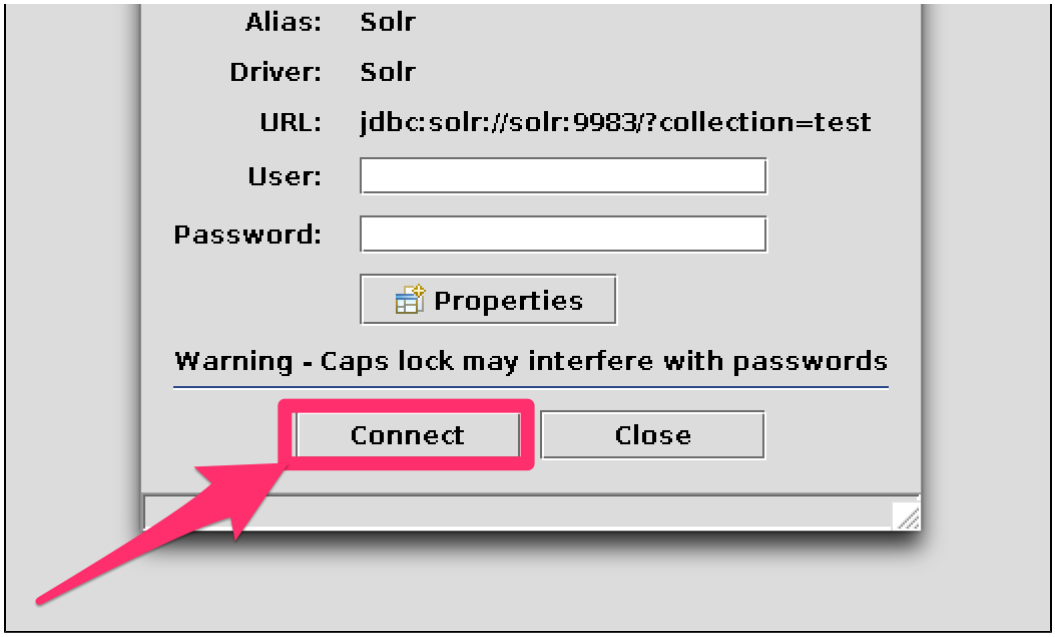


Configure the Alias



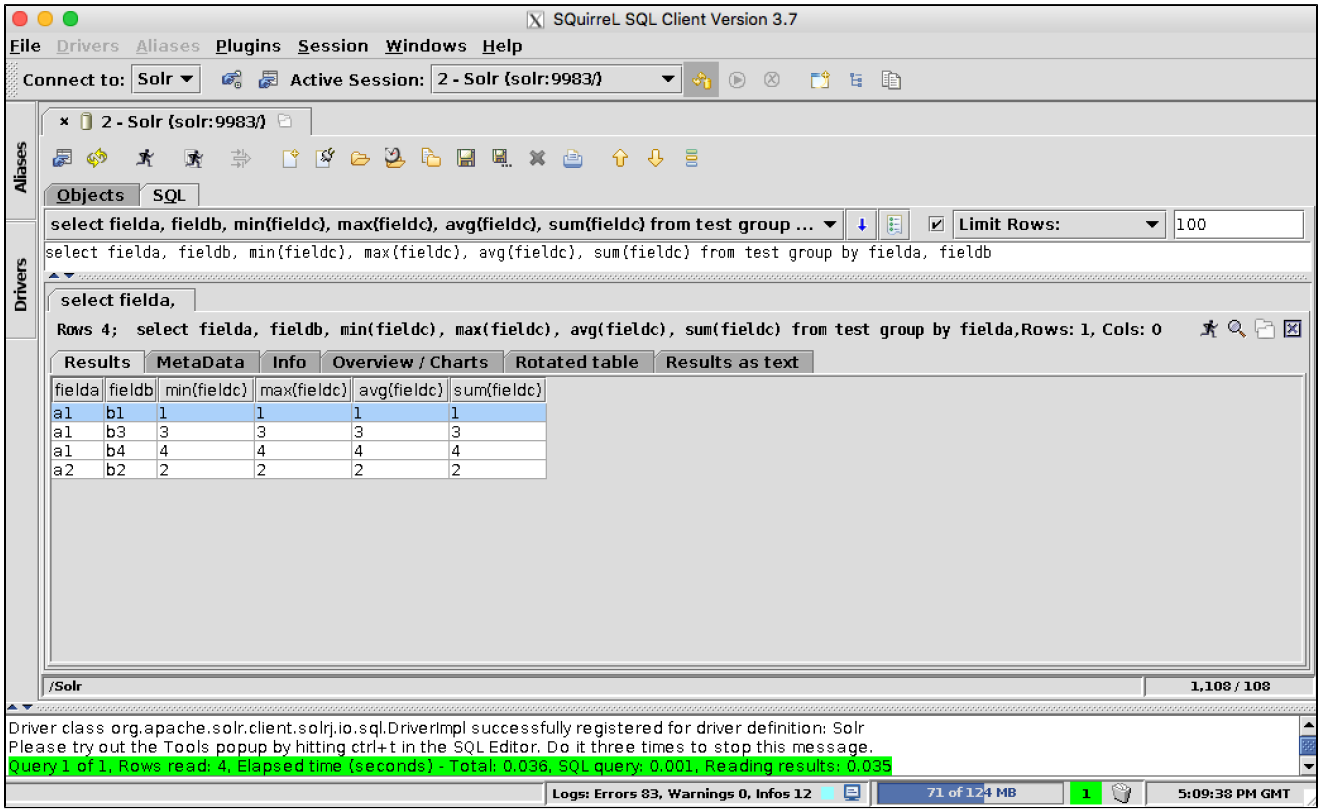
Connect to the Alias





Querying

Once you've successfully connected to Solr, you can use the SQL interface to enter queries and work with data.



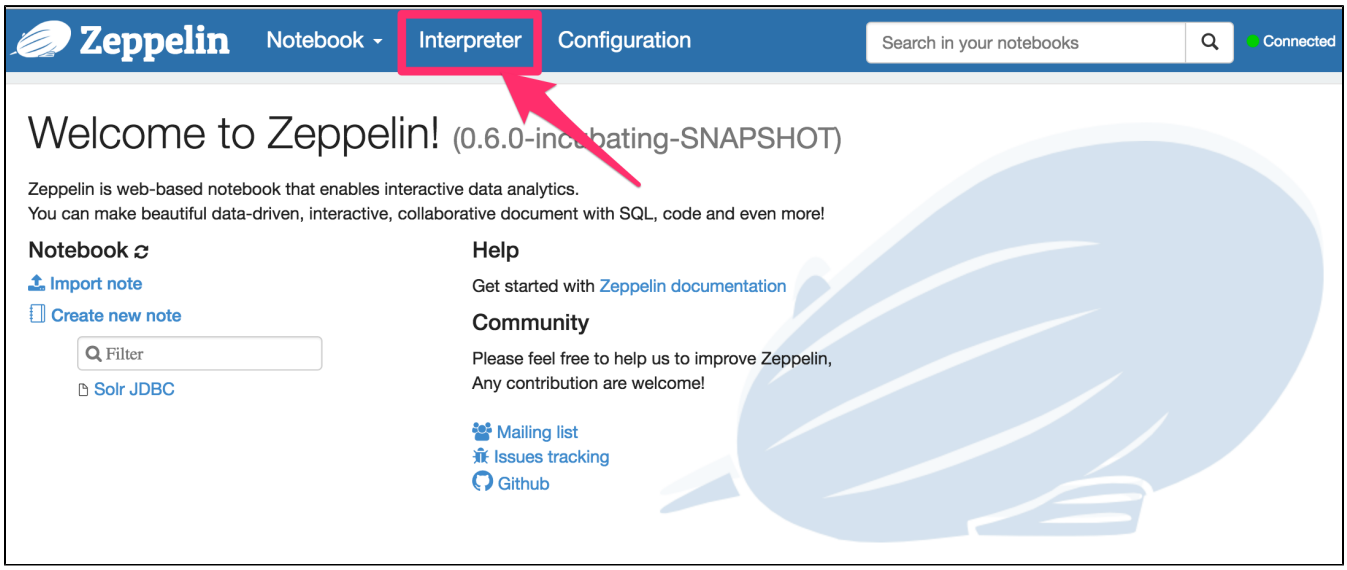
Solr JDBC - Apache Zeppelin

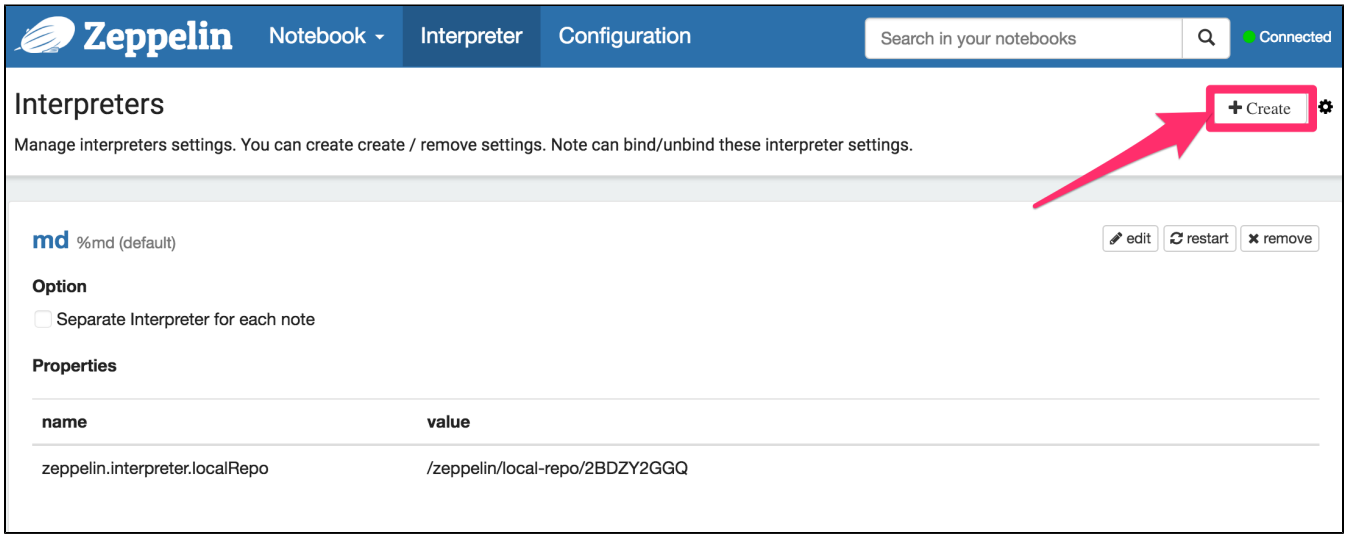
- Create the Apache Solr JDBC Interpreter
- Create a Notebook
- Query with the Notebook

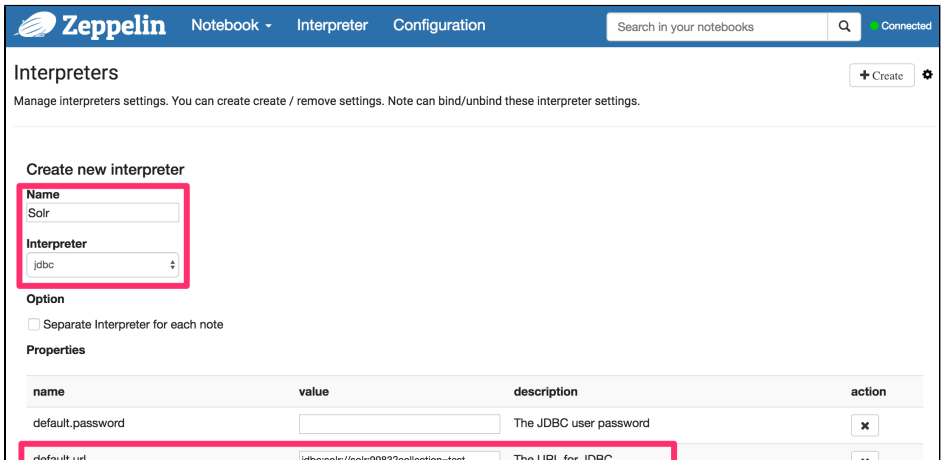
This requires Apache Zeppelin 0.6.0 or greater which contains the JDBC interpreter.

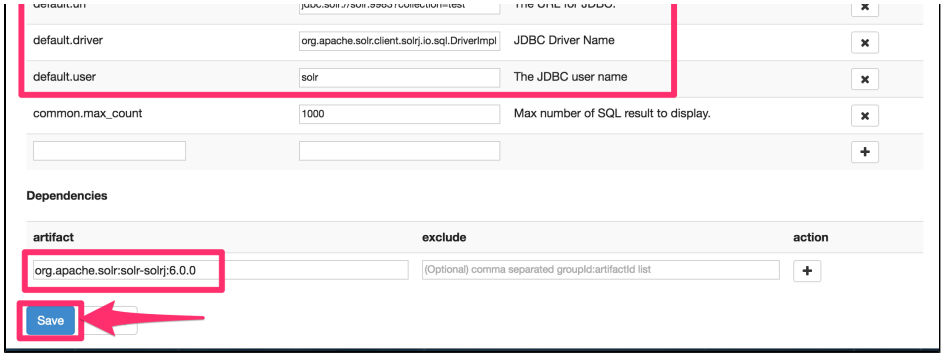
For Apache Zeppelin, you will need to create a JDBC interpreter for Solr. This will add SolrJ to the interpreter classpath. Once the interpreter has been created, you can create a notebook to issue queries.

Create the Apache Solr JDBC Interpreter

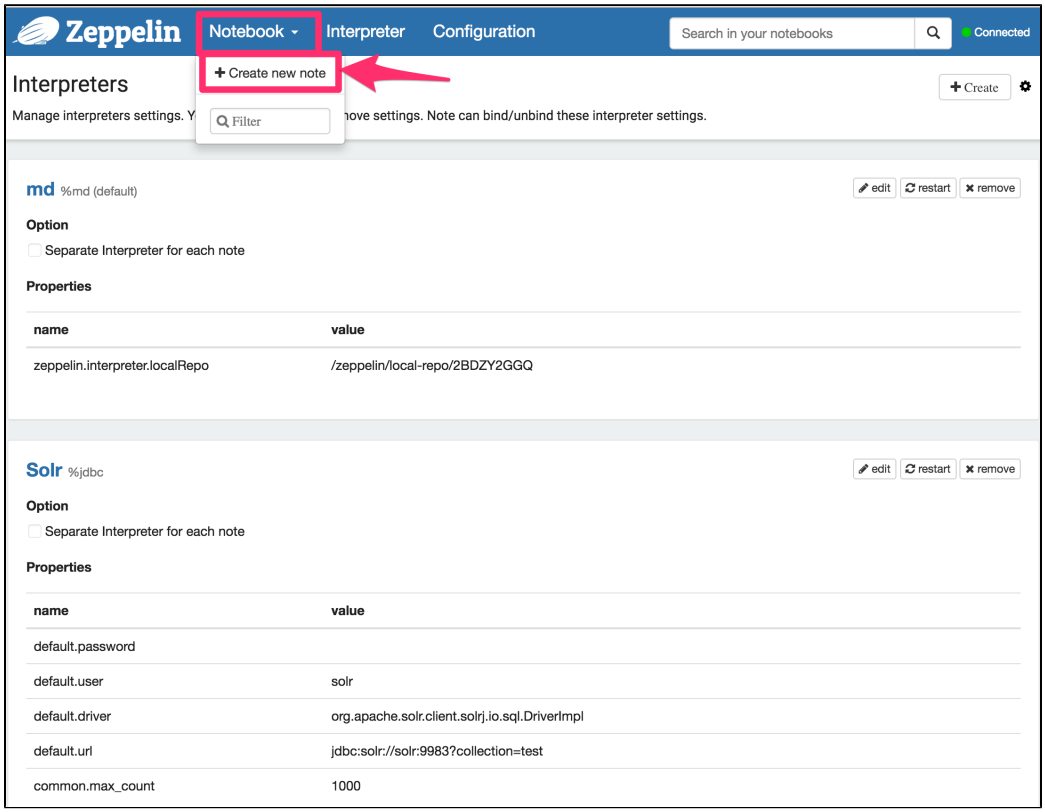


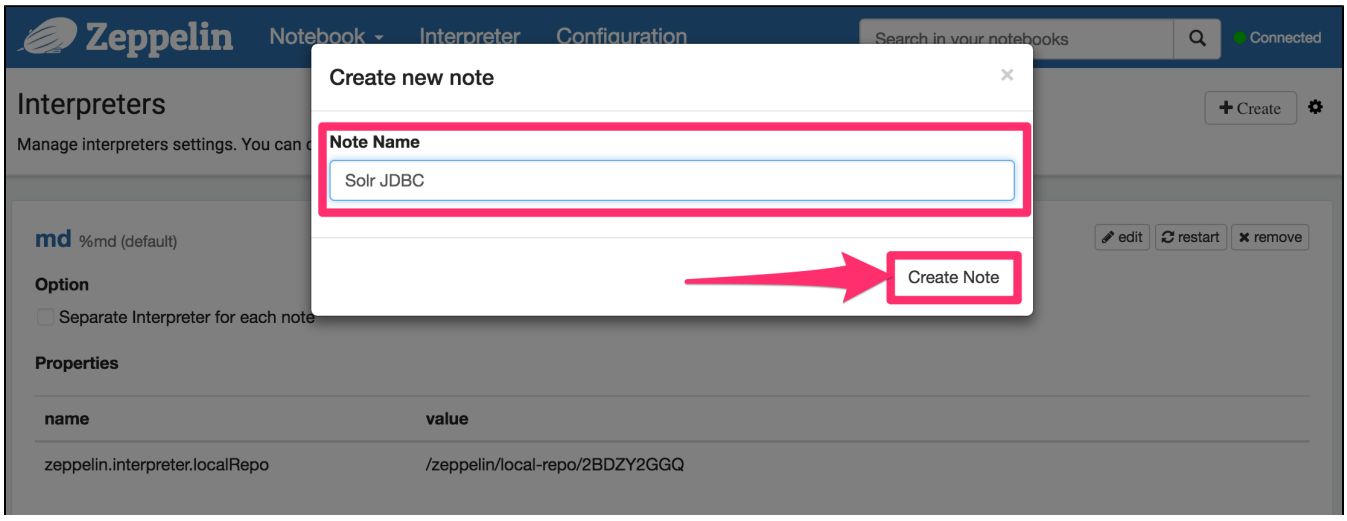






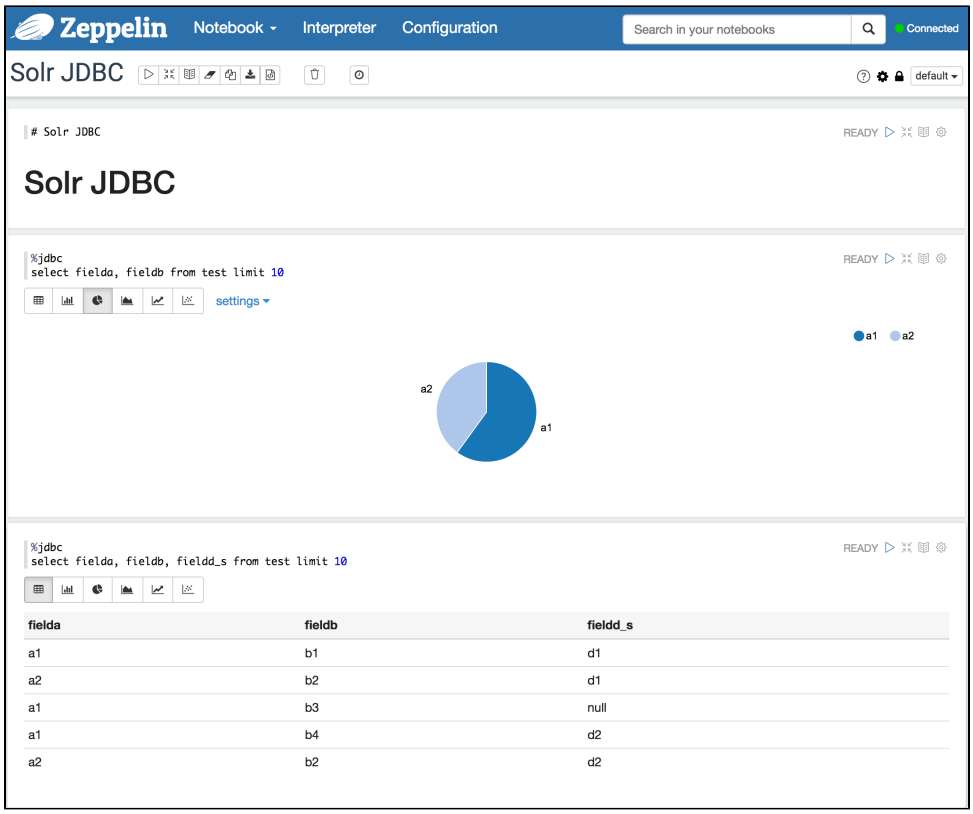
Create a Notebook







Query with the Notebook



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.

Configuration APIs: Describes several APIs used to configure Solr: Blob Store, Config, Request Parameters and Managed Resources.

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, either directly or via the Config API to create "Configuration Overlays" (`configoverlay.json`) to override the values in `solrconfig.xml`.

In `solrconfig.xml`, you configure important features such as:

- request handlers, which process the requests to Solr, such as requests to add documents to the index or requests to return results for a query
- 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 located in the `conf/` directory for each collection. Several well-commented example files can be found in the `server/solr/configsets/` directories demonstrating best practices for many different types of installations.

We've covered the options in the following sections:

- DataDir and DirectoryFactory in SolrConfig
- Lib Directives in SolrConfig
- Schema Factory Definition in SolrConfig
- IndexConfig in SolrConfig
- RequestHandlers and SearchComponents in SolrConfig
- InitParams in SolrConfig
- UpdateHandlers in SolrConfig
- Query Settings in SolrConfig
- RequestDispatcher in SolrConfig

- Update Request Processors
- Codec Factory

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. Of those below, strongly consider "config overlay" as the preferred approach, as it stays local to the config set and because it's easy to modify.

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 sample `solrconfig.xml` files, 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, you could override this using a JVM system property by launching the Solr it with:

```
bin/solr start -Dsolr.lock.type=none
```

In general, any Java system property that you want to set can be passed through the `bin/solr` script using the standard `-Dproperty=value` syntax. Alternatively, you can add common system properties to the `SOLR_OPTS` environment variable defined in the Solr include file (`bin/solr.in.sh`). For more information about how the Solr include file works, refer to: Taking Solr to Production.

Config API

The Config API allows you to use an API to modify Solr's configuration, specifically user defined properties. Changes made with this API are stored in a file named `configoverlay.json`. This file should only be edited with the API, but will look like this example:

```
{"userProps":{
    "dih.db.url":"jdbc:oracle:thin:@localhost:1521",
    "dih.db.user":"username",
    "dih.db.pass":"password"}}
```

For more details, see the section Config API.

`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 `conf/` directory of a collection using one of the example configurations, to override the lockType used.

```
#conf/solrcore.properties
solr.lock.type=none
```

Deprecation
solrcore.properties won't work in SolrCloud mode (it is not read from ZooKeeper). This feature is likely to be removed in the future. Instead, use another mechanism like a config overlay.

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`

Every Solr core has a core.properties file, automatically created when using the APIs. When you create a SolrCloud collection, you can pass through custom parameters to go into each core.properties that will be created, by prefixing the parameter name with "property." as a URL parameter. Example:

http://localhost:8983/solr/admin/collections?action=CREATE&name=gettingstarted&numShards=1&property.my.custom.prop=edismax

That would create a `core.properties` file that has these properties *and others omitted for brevity*:

```
#core.properties
name=gettingstarted
my.custom.prop=edismax
```

The `my.custom.prop` property can then be used as a variable, such as in `solrconfig.xml`:

```
<requestHandler name="/select">
  <lst name="defaults">
    <str name="defType">${my.custom.prop}</str>
  </lst>
</requestHandler>
```

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`

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 instance dir of the SolrCore. 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 `<dataDir>` 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"/>
```



If you are using Hadoop and would like to store your indexes in HDFS, you should use the `solr.HdfsDirectoryFactory` instead of either of the above implementations. For more details, see the section Running Solr on HDFS.

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" />
```

Schema Factory Definition in SolrConfig

Solr's Schema API enables remote clients to access Schema information, and make Schema modifications, through a REST interface. Other features such as Solr's Schemaless Mode also work via Schema modifications made programatically at run time.

While the "read" features of the Solr API are supported for all Schema types, support for making Schema modifications programatically depends on the `<schemaFactory/>` in use.

Managed Schema Default

When a `<schemaFactory/>` is not explicitly declared in a `solrconfig.xml` file, Solr implicitly uses a `ManagedIndexSchemaFactory`, which is by default `"mutable"` and keeps schema information in a `managed-schema` file.

```
<!-- An example of Solr's implicit default behavior if no
      no schemaFactory is explicitly defined.
 -->
  <schemaFactory class="ManagedIndexSchemaFactory">
    <bool name="mutable">true</bool>
    <str name="managedSchemaResourceName">managed-schema</str>
  </schemaFactory>
```

If you wish to explicitly configure `ManagedIndexSchemaFactory` the following options are available:

- `mutable` - controls whether changes may be made to the Schema data. This must be set to **true** to allow edits to be made with the Schema API.
- `managedSchemaResourceName` is an optional parameter that defaults to "managed-schema", and defines a new name for the schema file that can be anything other than "`schema.xml`".

With the default configuration shown above, you can use the Schema API to modify the schema as much as you want, and then later change the value of `mutable` to **false** if you wish to "lock" the schema in place and prevent future changes.

Classic `schema.xml`

An alternative to using a managed schema is to explicitly configure a `ClassicIndexSchemaFactory`. `Class`

`icIndexSchemaFactory` requires the use of a `schema.xml` configuration file, and disallows any programatic changes to the Schema at run time. The `schema.xml` file must be edited manually and is only loaded only when the collection is loaded.

```
<schemaFactory class="ClassicIndexSchemaFactory"/>
```

Switching from `schema.xml` to Managed Schema

If you have an existing Solr collection that uses `ClassicIndexSchemaFactory`, and you wish to convert to use a managed schema, you can simplify modify the `solrconfig.xml` to specify the use of the `ManagedIndexSchemaFactory`. Once Solr is restarted and it detects that a `schema.xml` file exists, but the `managedSchemaResourceName` file (ie: "`managed-schema`") does not exist, the existing `schema.xml` file will be renamed to `schema.xml.bak` and the contents are re-written to the managed schema file. 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 -->
```

You are now free to use the Schema API as much as you want to make changes, and remove the `schema.xml.bak`.

Changing to Manually Edited `schema.xml`

If you have started Solr with managed schema enabled and you would like to switch to manually editing a `schema.xml` file, you should take the following steps:

1. Rename the `managed-schema` file to `schema.xml`.
2. Modify `solrconfig.xml` to replace the `schemaFactory` class.
    a. Remove any `ManagedIndexSchemaFactory` definition if it exists.
    b. Add a `ClassicIndexSchemaFactory` definition as shown above
3. Reload the core(s).

If you are using SolrCloud, you may need to modify the files via ZooKeeper.

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>
```

Parameters covered in this section:

- Writing New Segments
- Merging Index Segments
- Compound File Segments
- Index Locks
- Other Indexing Settings

Writing New 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.

```
<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>
```

useCompoundFile

Controls whether newly written (and not yet merged) index segments should use the Compound File Segment format. The default is false.

```
<useCompoundFile>false</useCompoundFile>
```

Merging Index Segments

mergePolicyFactory

Defines how merging segments is done. The default in Solr is to use a `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.

```
<mergePolicyFactory class="org.apache.solr.index.TieredMergePolicyFactory">
  <int name="maxMergeAtOnce">10</int>
  <int name="segmentsPerTier">10</int>
</mergePolicyFactory>
```

Controlling Segment Sizes: Merge Factors

The most common adjustment some folks make to the configuration of TieredMergePolicy (or LogByteSizeMergePolicy) are the "merge factors" to change how many segments should be merged at one time. For TieredMergePolicy, this is controlled by setting the `<int name="maxMergeAtOnce">` and `<int name="segmentsPerTier">` options, while LogByteSizeMergePolicy has a single `<int name="mergeFactor">` option (all of which default to "`10`").

To understand why these options are important, consider what happens when an update is made to an index using LogByteSizeMergePolicy: Documents are always added to the most recently opened segment. When a segment fills up, 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 10, 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 10 of these larger segments, then they in turn are merged into an even larger single segment. This process can continue indefinitely.

When using TieredMergePolicy, the process is the same, but instead of a single `mergeFactor` value, the `segmentsPerTier` setting is used as the threshold to decide if a merge should happen, and the `maxMergeAtOnce` setting determines how many segments should be included in the merge.

Choosing the best merge factors 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.

Customizing Merge Policies

If the configuration options for the built-in merge policies do not fully suit your use case, you can customize them: either by creating a custom merge policy factory that you specify in your configuration, or by configuring a merge policy wrapper which uses a `wrapped.prefix` configuration option to control how the factory it wraps will be configured:

```
<mergePolicyFactory class="org.apache.solr.index.SortingMergePolicyFactory">
  <str name="sort">timestamp desc</str>
  <str name="wrapped.prefix">inner</str>
  <str name="inner.class">org.apache.solr.index.TieredMergePolicyFactory</str>
  <int name="inner.maxMergeAtOnce">10</int>
  <int name="inner.segmentsPerTier">10</int>
</mergePolicyFactory>
```

The example above shows Solr's `SortingMergePolicyFactory` being configured to sort documents in merged segments by `"timestamp desc"`, and wrapped around a `TieredMergePolicyFactory` configured to use the values `maxMergeAtOnce=10` and `segmentsPerTier=10` via the `inner` prefix defined by `SortingMergePolicyFactory`'s `wrapped.prefix` option. For more information on using `SortingMergePolicyFactory`, see the segmentTerminateEarly parameter.

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"/>
```

Compound File Segments

Each Lucene segment is typically comprised of a dozen or so files. Lucene can be configured to bundle all of the files for a segment into a single compound file using a file extension of `.cfs`; it's an abbreviation for Compound File Segment. CFS segments 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. On systems where the number of open files allowed per process is limited, CFS 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.

CFS: New Segments vs Merged Segments
To configure whether *newly written segments* should use CFS, see the `useCompoundFile` setting described above. To configure whether *merged segments* use CFS, review the Javadocs for your `mergePolicyFactory`.

Many Merge Policy implementations support `noCFSRatio` and `maxCFSSegmentSizeMB` settings with default values that prevent compound files from being used for large segments, but do use compound files for small segments.

Index Locks

lockType

The LockFactory options specify the locking implementation to use.

The set of valid lock type options depends on the DirectoryFactory you have configured. The values listed below are are supported by `StandardDirectoryFactory` (the default):

- `native` (default) uses NativeFSLockFactory to specify native OS file locking. If a second Solr process attempts to access the directory, it will fail. Do not use when multiple Solr web applications are attempting to share a single index.
- `simple` uses SimpleFSLockFactory to specify a plain file for locking.
- `single` (expert) uses SingleInstanceLockFactory. Use for special situations of a read-only index directory, or when there is no possibility of more than one process trying to modify the index (even sequentially). This type will protect against multiple cores within the *same* JVM attempting to access the same index. WARNING! If multiple Solr instances in different JVMs modify an index, this type will *not* protect against index corruption.
- `hdfs` uses HdfsLockFactory to support reading and writing index and transaction log files to a HDFS filesystem. See the section Running Solr on HDFS for more details on using this feature.

For more information on the nuances of each LockFactory, see http://wiki.apache.org/lucene-java/AvailableLockFactories.

```
<lockType>native</lockType>
```

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
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.

```
<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>
```

RequestHandlers and SearchComponents in SolrConfig

After the `<query>` section of `solrconfig.xml`, request handlers and search components are configured.

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.

These are often referred to as "requestHandler" and "searchComponent", which is how they are defined in `solrconfig.xml`.

Topics covered in this section:

- Request Handlers
  - SearchHandlers
  - UpdateRequestHandlers
  - ShardHandlers
  - Other Request Handlers
- Search Components
  - Default Components
  - First-Components and Last-Components
  - Components
  - Other Useful Components

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, typically as a path. For example, if Solr is installed at `http://localhost:8983/solr/` and you have a collection named "`gettingstarted`", you can make a request using URLs like this:

```
http://localhost:8983/solr/gettingstarted/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 (i.e., sending new documents to the index). By default, `/select` is a request handler that handles query requests.

Request handlers can also process requests for nested paths of their names, for example, a request using `/myhandler/extrapath` may be processed by a request handler registered with the name `/myhandler`. If a request handler is explicitly defined by the name `/myhandler/extrapath`, that would take precedence over the nested path. This assumes you are using the request handler classes included with Solr; if you create your own request handler, you should make sure it includes the ability to handle nested paths if you want to use them with your custom request handler.

It is also possible to configure defaults for request handlers with a section called `initParams`. These defaults can be used when you want to have common properties that will be used by each separate handler. For example, if you intend to create several request handlers that will all request the same list of fields in the response, you can configure an `initParams` section with your list of fields. For more information about `initParams`, see the section InitParams in SolrConfig.

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>
  </lst>
</requestHandler>
```

This example defines the `rows` parameter, which defines how many search results to return, to "10". 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

Search components define the logic that is used by the SearchHandler to perform queries for users.

Default Components

There are several default search components that work with all SearchHandlers without any additional configuration. If no components are defined (with the exception of `first-components` and `last-components` - see below), these are executed by default, in the following order:

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 .
expand	solr.ExpandComponent	Described in the section Collapse and Expand Results.

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`) the default components listed above.



`first-components` and/or `last-components` may only be used in conjunction with the default components. If you define your own `components`, the default components will not be executed, and `first-components` and `last-components` are disallowed.

```
<arr name="first-components">
  <str>mycomponent</str>
</arr>
<arr name="last-components">
  <str>spellcheck</str>
</arr>
```

Components

If you define `components`, the default components (see above) will not be executed, and `first-components` and `last-components` are disallowed:

```
<arr name="components">
  <str>mycomponent</str>
  <str>query</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.

InitParams in SolrConfig

An `<initParams>` section of `solrconfig.xml` allows you to define request handler parameters outside of the handler configuration.

The use cases are

- Some handlers are implicitly defined in code and there should be a way to add/append/override some of the implicitly defined properties
- There are a few properties that are used across handlers . This helps you keep only a single definition of those properties and apply them over multiple handlers

For example, if you want several of your search handlers to return the same list of fields, you can create an `<initParams>` section without having to define the same set of parameters in each request handler definition. If you have a single request handler that should return different fields, you can define the overriding parameters in individual `<requestHandler>` sections as usual.

The properties and configuration of an `<initParams>` section mirror the properties and configuration of a request handler. It can include sections for defaults, appends, and invariants, the same as any request handler.

For example, here is one of the `<initParams>` sections defined by default in the `data_driven_config` example:

```
<initParams path="/update/**,/query,/select,/tvrh,/elevate,/spell,/browse">
  <lst name="defaults">
    <str name="df">_text_</str>
  </lst>
</initParams>
```

This sets the default search field ("df") to be "_text_" for all of the request handlers named in the path section. If we later want to change the `/query` request handler to search a different field by default, we could override the `<initParams>` by defining the parameter in the `<requestHandler>` section for `/query`.

The syntax and semantics are similar to that of a `<requestHandler>` . The following are the attributes

property	Description
path	A comma-separated list of paths which will use the parameters. Wildcards can be used in paths to define nested paths, as described below.
name	The name of this set of parameters. The name can be directly in a requestHandler definition if a path is not explicitly named. If you give your `<initParams>` a name, you can refer to the params in a `<requestHandler>` that is not defined as a path. For example, if an `<initParams>` section has the name "myParams", you can call the name when defining your request handler: `<requestHandler name="/dump1" class="DumpRequestHandler" initParams="myParams"/>`

Wildcards

An `<initParams>` section can support wildcards to define nested paths that should use the parameters defined. A single asterisk (*) denotes that a nested path one level deeper should use the parameters. Double asterisks (**) denote all nested paths no matter how deep should use the parameters.

For example, if we have an `<initParams>` that looks like this:

```
<initParams name="myParams" path="/myhandler,/root/*,/root1/**">
  <lst name="defaults">
    <str name="fl">_text_</str>
  </lst>
  <lst name="invariants">
    <str name="rows">10</str>
  </lst>
  <lst name="appends">
    <str name="df">title</str>
  </lst>
</initParams>
```

We've defined three paths with this section:

- `/myhandler` declared as a direct path.
- `/root/*` with a single asterisk to indicate the parameters should apply to paths that are one level deep.
- `/root1/**` with double asterisks to indicate the parameters should apply to all nested paths, no matter how deep.

When we define the request handlers, the wildcards will work in the following ways:

```
<requestHandler name="/myhandler" class="SearchHandler"/>
```

The `/myhandler` class was named as a path in the `<initParams>` so this will use those parameters.

Next we have a request handler named `/root/search5`:

```
<requestHandler name="/root/search5" class="SearchHandler"/>
```

We defined a wildcard for nested paths that are one level deeper than `/root`, so this request handler will use the parameters. This one, however, will not, because `/root/search5/test` is more than one level deep from `/root`:

```
<requestHandler name="/root/search5/test" class="SearchHandler"/>
```

If we want to define all levels of nested paths, we should use double asterisks, as in the example path `/root1/**`:

```
<requestHandler name="/root1/search/tests" class="SearchHandler"/>
```

Any path under `/root1`, whether explicitly defined in a request handler or not, will use the parameters defined in the matching `initParams` section.

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
  - commit and softCommit
  - autoCommit
  - commitWithin
- Event Listeners
- Transaction Log

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`

In Solr, a `commit` is an action which asks Solr to "commit" those changes to the Lucene index files. By default commit actions result in a "hard commit" of all the Lucene index files to stable storage (disk). When a client includes a `commit=true` parameter with an update request, this ensures that all index segments affected by the adds & deletes on an update are written to disk as soon as index updates are completed.

If an additional flag `softCommit=true` is specified, then Solr performs a 'soft commit', meaning that Solr will commit your changes to the Lucene data structures quickly but not guarantee that the Lucene index files are written to 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.

Event Listeners

The UpdateHandler section is also where update-related event listeners can be configured. These can be triggered to occur after any commit (`event="postCommit"`) or only after optimize commands (`event="postOptimize"`).

Users can write custom update event listener classes, but a common use case is to run external executables via the `RunExecutableListener`:

Setting	Description
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.

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>
```

Three additional expert-level configuration settings affect indexing performance and how far a replica can fall behind on updates before it must enter into full recovery - see the section on write side fault tolerance for more information:

Setting Name	Type	Default	Description
numRecordsToKeep	int	100	The number of update records to keep per log
maxNumLogsToKeep	int	10	The maximum number of logs keep
numVersionBuckets	int	65536	The number of buckets used to keep track of max version values when checking for re-ordered updates; increase this value to reduce the cost of synchronizing access to version buckets during high-volume indexing, this requires (8 bytes (long) * numVersionBuckets) of heap space per Solr core.

An example, to be included under `<config><updateHandler>` in `solrconfig.xml`, employing the above advanced settings:

```
<updateLog>
  <str name="dir">${solr.ulog.dir:}</str>
  <int name="numRecordsToKeep">500</int>
  <int name="maxNumLogsToKeep">20</int>
  <int name="numVersionBuckets">65536</int>
</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
- Query Sizing and Warming
- Query-Related Listeners

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.

The most typical way Solr uses the `filterCache` is to cache results of each `fq` search parameter, though there are some other cases as well. Subsequent queries using the same parameter filter query result in cache hits and rapid returns of results. See Searching for a detailed discussion of the `fq` parameter. Another Solr feature using this cache is the `filter(...)` syntax in the default Lucene query parser.

Solr also uses 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.

The `queryResultCache` has an additional (optional) setting to limit the maximum amount of RAM used (`maxRamMB`). This lets you specify the maximum heap size, in megabytes, used by the contents of this cache. When the cache grows beyond this size, oldest accessed queries will be evicted until the heap usage of the cache decreases below the specified limit.

```
<queryResultCache class="solr.LRUCache"
                  size="512"
                  initialSize="512"
                  autowarmCount="128"
                  maxRamMB="1000"/>
```

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`. 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 (commented out) examples below can be found in the `solrconfig.xml` file of the `sample_techproducts_configs` config set included with Solr, and demonstrate using the `solr.QuerySenderListener` class to warm a set of explicit queries:

```
<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 comes from a *sample* `solrconfig.xml`. A key best practice is to modify these defaults before taking your application to production, but please note: while the sample queries are commented out in the section for the "newSearcher", the sample quer 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 HTTP `RequestDispatcher` implementation responds to 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
- requestParsers Element
- httpCaching Element

`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 used by any Solr component, 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.

Update Request Processors

Every update request received by Solr is run through a chain of plugins known as Update Request Processor. This can be useful, for example, to add a field to the document being indexed or to change the value of a particular field or to drop an update if the incoming document doesn't fulfill certain criteria. In fact, a surprisingly large number of features in Solr are implemented as Update Processors and therefore it is necessary to understand how such plugins work and where are they configured.

Topics in this section:

- Anatomy and life cycle
- Configuration
- Update processors in SolrCloud
- Using custom chains
- Update Request Processor Factories

Anatomy and life cycle

An Update Request Processor is created as part of a chain of one or more update processors. Solr creates a default update request processor chain comprising of a few update request processors which enable essential Solr features. This default chain is used to process every update request unless a user chooses to configure and

specify a different custom update request processor chain.

The easiest way to describe an Update Request Processor is to look at the Javadocs of the abstract class UpdateRequestProcessor. Every UpdateRequestProcessor must have a corresponding factory class which extends UpdateRequestProcessorFactory. This factory class is used by Solr to create a new instance of this plugin. Such a design provides two benefits:

1. An update request processor need not be thread safe because it is used by one and only one request thread and destroyed once the request is complete.
2. The factory class can accept configuration parameters and maintain any state that may be required between requests. The factory class must be thread-safe.

Every update request processor chain is constructed during loading of a Solr core and cached until the core is unloaded. Each UpdateRequestProcessorFactory specified in the chain is also instantiated and initialized with configuration that may have been specified in solrconfig.xml.

When an update request is received by Solr, it looks up the update chain to be used for this request. A new instance of each UpdateRequestProcessor specified in the chain is created using the corresponding factory. The update request is parsed into corresponding UpdateCommand objects which are run through the chain. Each UpdateRequestProcessor instance is responsible for invoking the next plugin in the chain. It can choose to short circuit the chain by not invoking the next processor and even abort further processing by throwing an exception.



A single update request may contain a batch of multiple new documents or deletes and therefore the corresponding processXXX methods of an UpdateRequestProcessor will be invoked multiple times for every individual update. However, it is guaranteed that a single thread will serially invoke these methods.

Configuration

Update request processors chains can be created by either creating the whole chain directly in solrconfig.xml or by creating individual update processors in solrconfig.xml and then dynamically creating the chain at run-time by specifying all processors via request parameters.

However, before we understand how to configure update processor chains, we must learn about the default update processor chain because it provides essential features which are needed in most custom request processor chains as well.

The default update request processor chain

In case no update processor chains are configured in solrconfig.xml, Solr will automatically create a default update processor chain which will be used for all update requests. This default update processor chain consists of the following processors (in order):

1. LogUpdateProcessorFactory - Tracks the commands processed during this request and logs them
2. DistributedUpdateProcessorFactory - Responsible for distributing update requests to the right node e.g. routing requests to the leader of the right shard and distributing updates from the leader to each replica. This processor is activated only in SolrCloud mode.
3. RunUpdateProcessorFactory - Executes the update using internal Solr APIs.

Each of these perform an essential function and as such any custom chain usually contain all of these processors. The RunUpdateProcessorFactory is usually the last update processor in any custom chain.

Custom update request processor chain

The following example demonstrates how a custom chain can be configured inside solrconfig.xml.

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>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

In the above example, a new update processor chain named "dedupe" is created with SignatureUpdateProcessorFactory, LogUpdateProcessorFactory and RunUpdateProcessorFactory in the chain. The SignatureUpdateProcessorFactory is further configured with different parameters such as "signatureField", "overwriteDupes" etc. This chain is an example of how Solr can be configured to perform de-duplication of documents by calculating a signature using the value of name, features, cat fields which is then used as the "id" field. As you may have noticed, this chain does not specify the DistributedUpdateProcessorFactory - because this processor is critical for Solr to operate properly, Solr will automatically insert DistributedUpdateProcessorFactory in this chain that does not include it just prior to the RunUpdateProcessorFactory.

RunUpdateProcessorFactory
Do not forget to add RunUpdateProcessorFactory at the end of any chains you define in solrconfig.xml otherwise update requests processed by that chain will not actually affect the indexed data.

Configuring individual processors as top-level plugins

Update request processors can also be configured independent of a chain in solrconfig.xml.

updateProcessor

```
<updateProcessor class="solr.processor.SignatureUpdateProcessorFactory"
name="signature">
  <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>
</updateProcessor>
<updateProcessor class="solr.RemoveBlankFieldUpdateProcessorFactory"
name="remove_blanks"/>
```

In this case, an instance of SignatureUpdateProcessorFactory is configured with the name "signature" and a RemoveBlankFieldUpdateProcessorFactory is defined with the name "remove_blanks". Once the above has been specified in solrconfig.xml, we can be refer to them in update request processor chains in solrconfig.xml as follows:

updateRequestProcessorChains and updateProcessors

```
<updateProcessorChain name="custom" processor="remove_blanks,signature">
  <processor class="solr.RunUpdateProcessorFactory" />
</updateProcessorChain>
```

Update processors in SolrCloud

In a single node, stand alone Solr, each update is run through all the update processors in a chain exactly once. But the behavior of update request processors in SolrCloud deserves special consideration.

A critical SolrCloud functionality is the routing and distributing of requests – for update requests this routing is implemented by the DistributedUpdateRequestProcessor, and this processor is given a special status by Solr due to its important function.

In a distributed SolrCloud situation setup, All processors in the chain *before* the DistributedUpdateProcessor are run on the first node that receives an update from the client, regardless of this nodes status as a leader or replica. The DistributedUpdateProcessor then forwards the update to the appropriate shard leader for the update (or to multiple leaders in the event of an update that affects multiple documents, such as a delete by query, or commit). The shard leader uses a transaction log to apply Atomic Updates & Optimistic Concurrency a nd then forwards the update to all of the shard replicas. The leader and each replica run all of the processors in the chain that are listed *after* the DistributedUpdateProcessor.

For example, consider the "dedupe" chain which we saw in a section above. Assume that a 3 node SolrCloud cluster exists where node A hosts the leader of shard1, node B hosts the leader of shard2 and node C hosts the replica of shard2. Assume that an update request is sent to node A which forwards the update to node B (because the update belongs to shard2) which then distributes the update to its replica node C. Let's see what happens at each node:

- **Node A**: Runs the update through the SignatureUpdateProcessor (which computes the signature and puts it in the "id" field), then LogUpdateProcessor and then DistributedUpdateProcessor. This processor determines that the update actually belongs to node B and is forwarded to node B. The update is not processed further. This is required because the next processor which is RunUpdateProcessor will execute the update against the local shard1 index which would lead to duplicate data on shard1 and shard2.
- **Node B**: Receives the update and sees that it was forwarded by another node. The update is directly sent to DistributedUpdateProcessor because it has already been through the SignatureUpdateProcessor on node A and doing the same signature computation again would be redundant. The DistributedUpdateProc essor determines that the update indeed belongs to this node, distributes it to its replica on Node C and then forwards the update further in the chain to RunUpdateProcessor.
- **Node C**: Receives the update and sees that it was distributed by its leader. The update is directly sent to DistributedUpdateProcessor which performs some consistency checks and forwards the update further in the chain to RunUpdateProcessor.

In summary:

1. All processors before DistributedUpdateProcessor are only run on the first node that receives an update request whether it be a forwarding node (e.g. node A in the above example) or a leader (e.g. node B). We call these pre-processors or just processors.
2. All processors after DistributedUpdateProcessor run only on the leader and the replica nodes. They are not executed on forwarding nodes. Such processors are called "post-processors".

In the previous section, we saw that the updateRequestProcessorChain was configured with `processor="rem ove_blanks, signature"`. This means that such processors are of the #1 kind and are run only on the forwarding nodes. Similarly, we can configure them as the #2 kind by specifying with the attribute "post-processor" as follows:

post-processors

```
<updateProcessorChain name="custom" processor="signature"
post-processor="remove_blanks">
  <processor class="solr.RunUpdateProcessorFactory" />
</updateProcessorChain>
```

However executing a processor only on the forwarding nodes is a great way of distributing an expensive computation such as de-duplication across a SolrCloud cluster by sending requests randomly via a load balancer. Otherwise the expensive computation is repeated on both the leader and replica nodes.

Pre-processors and Atomic Updates
Because DistributedUpdateProcessor is responsible for processing Atomic Updates into full documents on the leader node, this means that pre-processors which are executed only on the forwarding nodes can only operate on the partial document. If you have a processor which must process a full document then the only choice is to specify it as a post-processor.

Using custom chains

update.chain request parameter

The update.chain parameter can be used in any update request to choose a custom chain which has been configured in solrconfig.xml. For example, in order to choose the "dedupe" chain described in a previous section, one can issue the following request:

update.chain

```
curl
"http://localhost:8983/solr/gettingstarted/update/json?update.chain=dedupe&commit=tr
ue" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  }
]'
```

The above should dedupe the two identical documents and index only one of them.

processor & post-processor request parameters

We can dynamically construct a custom update request processor chain using the "processor" and "post-processor" request parameters. Multiple processors can be specified as a comma-separated value for these two parameters. For example:

Constructing a chain at request time

```
# Executing processors configured in solrconfig.xml as (pre)-processors
curl
"http://localhost:8983/solr/gettingstarted/update/json?processor=remove_blanks,signa
ture&commit=true" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]

  }
]'

# Executing processors configured in solrconfig.xml as pre and post processors
curl
"http://localhost:8983/solr/gettingstarted/update/json?processor=remove_blanks&post-
processor=signature&commit=true" -H 'Content-type: application/json' -d '
[
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  },
  {
    "name" : "The Lightning Thief",
    "features" : "This is just a test",
    "cat" : ["book","hardcover"]
  }
]'
```

In the first example, Solr will dynamically create a chain which has "signature" and "remove_blanks" as pre-processors to be executed only on the forwarding node where as in the second example, "remove_blanks" will be executed as a pre-processor and "signature" will be executed on the leader and replicas as a post processor.

Configuring a custom chain as a default

We can also specify a custom chain to be used by default for all requests sent to specific update handlers instead of specifying the names in request parameters for each request.

This can be done by adding either "update.chain" or "processor" and "post-processor" as default parameter for a given path which can be done either via InitParams in SolrConfig or by adding them in a "defaults" section which is supported by all request handlers.

The following is an actual InitParam defined in the schemaless configuration which applies a custom update chain to all request handlers starting with "/update/".

InitParams

```
<initParams path="/update/**">
  <lst name="defaults">
    <str name="update.chain">add-unknown-fields-to-the-schema</str>
  </lst>
</initParams>
```

Alternately, one can achieve a similar effect using the "defaults" as shown in the example below:

defaults

```
<requestHandler name="/update/extract"
                startup="lazy"
                class="solr.extraction.ExtractingRequestHandler" >
  <lst name="defaults">
    <str name="update.chain">add-unknown-fields-to-the-schema</str>
  </lst>
</requestHandler>
```

Update Request Processor Factories

What follows are brief descriptions of the currently available update request processors. UpdateRequestProcessorFactories can be integrated into an update chain in solrconfig.xml as necessary. You are strongly urged to examine the Javadocs for these classes; these descriptions are abridged snippets taken for the most part from the Javadocs.

- **AddSchemaFieldsUpdateProcessorFactory:** This processor will dynamically add fields to the schema if an input document contains one or more fields that don't match any field or dynamic field in the schema.
- **CloneFieldUpdateProcessorFactory:** Clones the values found in any matching *source* field into the configured *dest* field.
- **DefaultValueUpdateProcessorFactory:** A simple processor that adds a default value to any document which does not already have a value in fieldName.
- **DocBasedVersionConstraintsProcessorFactory:** This Factory generates an UpdateProcessor that helps to enforce version constraints on documents based on per-document version numbers using a configured name of a versionField.
- **DocExpirationUpdateProcessorFactory:** Update Processor Factory for managing automatic "expiration" of documents.
- **IgnoreCommitOptimizeUpdateProcessorFactory:** Allows you to ignore commit and/or optimize requests from client applications when running in SolrCloud mode, for more information, see: Shards and Indexing Data in SolrCloud
- **RegexpBoostProcessorFactory:** A processor which will match content of "inputField" against regular expressions found in "boostFilename", and if it matches will return the corresponding boost value from the file and output this to "boostField" as a double value.
- **SignatureUpdateProcessorFactory:** Uses a defined set of fields to generate a hash "signature" for the document. Useful for only indexing one copy of "similar" documents.
- **StatelessScriptUpdateProcessorFactory:** An update request processor factory that enables the use of update processors implemented as scripts.
- **TimestampUpdateProcessorFactory:** An update processor that adds a newly generated date value of "NOW" to any document being added that does not already have a value in the specified field.
- **URLClassifyProcessorFactory:** Update processor which examines a URL and outputs to various other fields with characteristics of that URL, including length, number of path levels, whether it is a top level URL (levels==0), whether it looks like a landing/index page, a canonical representation of the URL (e.g.

stripping index.html), the domain and path parts of the URL etc.
- **UUIDUpdateProcessorFactory:** An update processor that adds a newly generated UUID value to any document being added that does not already have a value in the specified field.

FieldMutatingUpdateProcessorFactory derived factories

These factories all provide functionality to *modify* fields in a document as they're being indexed. When using any of these factories, please consult the FieldMutatingUpdateProcessorFactory javadocs for details on the common options they all support for configuring which fields are modified.

- **ConcatFieldUpdateProcessorFactory:** Concatenates multiple values for fields matching the specified conditions using a configurable delimiter.
- **CountFieldValuesUpdateProcessorFactory:** Replaces any list of values for a field matching the specified conditions with the the count of the number of values for that field.
- **FieldLengthUpdateProcessorFactory:** Replaces any CharSequence values found in fields matching the specified conditions with the lengths of those CharSequences (as an Integer).
- **FirstFieldValueUpdateProcessorFactory:** Keeps only the first value of fields matching the specified conditions.
- **HTMLStripFieldUpdateProcessorFactory** : Strips all HTML Markup in any CharSequence values found in fields matching the specified conditions.
- **IgnoreFieldUpdateProcessorFactory:** Ignores and removes fields matching the specified conditions from any document being added to the index.
- **LastFieldValueUpdateProcessorFactory:** Keeps only the last value of fields matching the specified conditions.
- **MaxFieldValueUpdateProcessorFactory:** An update processor that keeps only the the maximum value from any selected fields where multiple values are found.
- **MinFieldValueUpdateProcessorFactory:** An update processor that keeps only the the minimum value from any selected fields where multiple values are found.
- **ParseBooleanFieldUpdateProcessorFactory:** Attempts to mutate selected fields that have only CharSequence-typed values into Boolean values.
- **ParseDateFieldUpdateProcessorFactory:** Attempts to mutate selected fields that have only CharSequence-typed values into Solr date values.
- **ParseNumericFieldUpdateProcessorFactory derived classes:**
  - **ParseDoubleFieldUpdateProcessorFactory:** Attempts to mutate selected fields that have only CharSequence-typed values into Double values.
  - **ParseFloatFieldUpdateProcessorFactory :** Attempts to mutate selected fields that have only CharSequence-typed values into Float values.
  - **ParseIntFieldUpdateProcessorFactory :** Attempts to mutate selected fields that have only CharSequence-typed values into Integer values.
  - **ParseLongFieldUpdateProcessorFactory :** Attempts to mutate selected fields that have only CharSequence-typed values into Long values.
- **PreAnalyzedUpdateProcessorFactory :** An update processor that parses configured fields of any document being added using *PreAnalyzedField* with the configured format parser.
- **RegexReplaceProcessorFactory :** An updated processor that applies a configured regex to any CharSequence values found in the selected fields, and replaces any matches with the configured replacement string.
- **RemoveBlankFieldUpdateProcessorFactory :** Removes any values found which are CharSequence with a length of 0. (ie: empty strings).
- **TrimFieldUpdateProcessorFactory:** Trims leading and trailing whitespace from any CharSequence values found in fields matching the specified conditions.
- **TruncateFieldUpdateProcessorFactory:** Truncates any CharSequence values found in fields matching the specified conditions to a maximum character length.
- **UniqFieldsUpdateProcessorFactory :** Removes duplicate values found in fields matching the specified conditions.

Update Processor factories that can be loaded as plugins

These processors are included in Solr releases as "contribs", and require additional jars loaded at runtime. See

the README files associated with each contrib for details:

- The `langid` contrib provides**:**
  - **LangDetectLanguageIdentifierUpdateProcessorFactory :** Identifies the language of a set of input fields using http://code.google.com/p/language-detection
  - **TikaLanguageIdentifierUpdateProcessorFactory :** Identifies the language of a set of input fields using Tika's LanguageIdentifier.
- The `uima` contrib provides:
  - **UIMAUpdateRequestProcessorFactory :** Update document(s) to be indexed with UIMA extracted information.

Update Processor factories you should not modify or remove.

These are listed for completeness, but are part of the Solr infrastructure, particularly SolrCloud. Other than insuring you do *not* remove them when modifying the update request handlers (or any copies you make), you will rarely, if ever, need to change these.

- **DistributedUpdateProcessorFactory:** Used to distribute updates to all necessary nodes.
  - **NoOpDistributingUpdateProcessorFactory:** An alternative No-Op implementation of DistributingUpdateProcessorFactory that always returns null. Designed for experts who want to bypass distributed updates and use their own custom update logic.
- **LogUpdateProcessorFactory:** A logging processor. This keeps track of all commands that have passed through the chain and prints them on finish().
- **RunUpdateProcessorFactory :** Executes the update commands using the underlying UpdateHandler. Almost all processor chains should end with an instance of *RunUpdateProcessorFactory* unless the user is explicitly executing the update commands in an alternative custom *UpdateRequestProcessorFactory*.

Codec Factory

A `codecFactory` can be specified in `solrconfig.xml` to determine which Lucene `Codec` is used when writing the index to disk.

If not specified, Lucene's default codec is implicitly used, but a `solr.SchemaCodecFactory` is also available which supports 2 key features:

- Schema based per-fieldtype configuration for `docValuesFormat` and `postingsFormat` - see the Field Type Definitions and Properties section for more details.
- A `compressionMode` option:
  - `BEST_SPEED` (default) is optimized for search speed performance
  - `BEST_COMPRESSION` is optimized for disk space usage

Example:

```
<codecFactory class="solr.SchemaCodecFactory">
  <str name="compressionMode">BEST_COMPRESSION</str>
</codecFactory>
```

Solr Cores and solr.xml

In Solr, the term *core* is used to refer to a single index and associated transaction log and configuration files (including the `solrconfig.xml` and Schema files, among others). Your Solr installation can have multiple

cores if needed, which allows you to index data with different structures in the same server, and maintain more control over how your data is presented to different audiences. In SolrCloud mode you will be more familiar with the term *collection.* Behind the scenes a collection consists of one or more cores.

Cores can be created using `bin/solr` script or as part of SolrCloud collection creation using the APIs. Core-specific properties (such as the directories to use for the indexes or configuration files, the core name, and other options) are defined in a `core.properties` file. Any `core.properties` file in any directory of your Solr installation (or in a directory under where `solr_home` is defined) will be found by Solr and the defined properties will be used for the core named in the file.

In standalone mode, `solr.xml` must reside in `solr_home`. In SolrCloud mode, `solr.xml` will be loaded from Zookeeper if it exists, with fallback to `solr_home`.



In older versions of Solr, cores had to be predefined as `<core>` tags in `solr.xml` in order for Solr to know about them. Now, however, Solr supports automatic discovery of cores and they no longer need to be explicitly defined. The recommended way is to dynamically create cores/collections using the APIs.

The following sections describe these options in more detail.

- **Format of solr.xml**: Details on how to define `solr.xml`, including the acceptable parameters for the `solr.xml` file
- **Defining core.properties**: Details on placement of `core.properties` and available property options.
- **CoreAdmin API**: Tools and commands for core administration using a REST API.
- **Config Sets**: How to use configsets to avoid duplicating effort when defining a new core.

Format of solr.xml

This section will describe the default `solr.xml` file included with Solr and how to modify it for your needs. For details on how to configure `core.properties`, see the section Defining core.properties.

- Defining solr.xml
  - Solr.xml Parameters
    - The <solr> Element
    - The <solrcloud> element
    - The <logging> element
      - The <logging><watcher> element
    - The <shardHandlerFactory> element
- Substituting JVM System Properties in solr.xml

Defining solr.xml

You can find `solr.xml` in your Solr Home directory or in Zookeeper. The default `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.

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`.

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, `<str name="adminHandler">com.myorg.MyAdminHandler</str>` 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`	Currently non-operational.
`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. Custom handlers may be placed in this directory

`shareSchema`	This attribute, when set to true, ensures that the multiple cores pointing to the same Schema resource file 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 file.
`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 url context path.
`hostPort`	The port Solr uses to access cores. In the default `solr.xml` file, this is set to `${jetty.port:8983}`, which will use the Solr port defined in Jetty, and otherwise fall back to 8983.
`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 (ms) 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.
`zkCredentialsProvider` & `zkACLProvider`	Optional parameters that can be specified if you are using ZooKeeper Access Control.

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">
```

Since this is a custom shard handler, sub-elements are specific to the implementation. The default and only shard handler provided by Solr is the HttpShardHandlerFactory in which case, the following sub-elements can be specified:

Node	Description
socketTimeout	The read timeout for intra-cluster query and administrative requests. The default is the same as the distribUpdateSoTimeout specified in the solrcloud section.
connTimeout	The connection timeout for intra-cluster query and administrative requests. Defaults to the distribUpdateConnTimeout specified in the solrcloud section
urlScheme	URL scheme to be used in distributed search
maxConnectionsPerHost	Maximum connections allowed per host. Defaults to 20
maxConnections	Maximum total connections allowed. Defaults to 10000
corePoolSize	The initial core size of the threadpool servicing requests. Default is 0.
maximumPoolSize	The maximum size of the threadpool servicing requests. Default is unlimited.
maxThreadIdleTime	The amount of time in seconds that idle threads persist for in the queue, before being killed. Default is 5 seconds.
sizeOfQueue	If the threadpool uses a backing queue, what is its maximum size to use direct handoff. Default is to use a SynchronousQueue.
fairnessPolicy	A boolean to configure if the threadpool favours fairness over throughput. Default is false to favor throughput.

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, the `socketTimeout` and `connTimeout` values are each set to "0". However, if you start Solr using '`bin/solr -DsocketTimeout=1000`', the `socketTimeout` option of the `HttpShardHandlerFactory` to be overridden using a value of 1000ms, while 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>
```

Maximum connections allowed per host

Defining core.properties

Core discovery means that creating a core is as simple as a `core.properties` file located on disk. 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.

A minimal `core.properties` file looks like this (however, it can also be empty, see information on placement of `core.properties` below):

```
name=my_core_name
```

Placement of core.properties

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
```

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 Solr.

Your `core.properties` file can be empty if necessary. Suppose `core.properties` is located in `./cores/core1` (relative to `solr_home`) but is empty. In that case, the 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.

Defining core.properties Files

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:

Property	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` but please note that if you are using a "managed schema" (the default behavior) then any value for this property which does not match the effective `managedSchemaResourceName` will be read once, backed up, and converted for managed schema use. See Schema Factory Definition in SolrConfig for details.
`dataDir`	The core's data directory (where indexes are stored) as either an absolute pathname, or a path relative to the value of `instanceDir`. This is `data` by default.
`configSet`	The name of a defined configset, if desired, to use to configure the core (see the Config Sets for more details).
`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`	Used only in SolrCloud, this is a unique identifier for the node hosting this replica. By default a coreNodeName is generated automatically, but setting this attribute explicitly allows you to manually assign a new core to replace an existing replica. For example: when replacing a machine that has had a hardware failure by restoring from backups on a new machine with a new hostname or port..
`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. For more information on how to define local properties, see the section Substituting Properties in Solr Config Files.

CoreAdmin API

The Core Admin API is primarily used under the covers by the Collections API when running a SolrCloud cluster. SolrCloud users should not typically use the CoreAdmin API directly – but it may be useful for users of single-node or master/slave Solr installations for core maintenance operations.

The CoreAdmin API is implemented by the CoreAdminHandler, which is a special purpose SolrRequestHandler that is used to manage Solr cores. Unlike normal SolrRequestHandlers, the CoreAdminHandler is not attached to a single core. Instead, it there is a single instance of the CoreAdminHandler in each Solr node that manages all the cores running in that node and is accessible at the `/solr/admin/cores` path.

CoreAdmin actions can be executed by via HTTP requests that specify an "`action`" request parameter, with additional action specific arguments 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 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/dir&config=config_file_name.xml&dataDir=data
```

CREATE must be able to find a configuration!
Your CREATE call must be able to find a configuration, or it will not succeed.

When you are running SolrCloud and create a new core for a collection, the configuration will be inherited from the collection – each collection is linked to a configName, which is stored in the zookeeper database. This satisfies the config requirement. There is something to note, though – if you're running SolrCloud, you should **NOT** be using the CoreAdmin API at all. Use the Collections API.

When you are not running SolrCloud, if you have Config Sets defined, you can use the configSet parameter as documented below. If there are no config sets, then the instanceDir specified in the CREATE call must already exist, and it must contain a conf directory which in turn must contain `solrconfig.xml` and your schema, which is usually named either `managed-schema` or `schema.xml`, as well as any files referenced by those configs. The config and schema filenames could be specified with the config and schema parameters, but these are expert options. One thing you COULD do to avoid creating the conf directory is use config and schema parameters that point at absolute paths, but this can lead to confusing configurations unless you fully understand what you are doing.

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	No	whatever is specified for "name" parameter	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 (i.e., `solrconfig.xml`) relative to `instanceDir`.
schema	string	No		Name of the schema file to use for the core. Please note that if you are using a "managed schema" (the default behavior) then any value for this property which does not match the effective `managedSchemaResourceName` will be read once, backed up, and converted for managed schema use. See Schema Factory Definition in SolrConfig for details.
dataDir	string	No		Name of the data directory relative to `instanceDir`.

configSet	string	No		Name of the configset to use for this core. For more information, see the section 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 the section on defining 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=my_core&collection=my_collection&shard=shard2
```

While it's possible to create a core for a non-existent collection, this approach is not supported and not recommended. Always create a collection using the Collections API before creating a core directly for it.

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.

```
http://localhost:8983/solr/admin/cores?action=RELOAD&core=core0
```

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.

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=new_core_name&indexDir=/solr_home/core1/data/index&indexDir=/solr_home/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=new_core_name&srcCore=core1&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/in
dex/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
&split.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
&targetCore=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
   /managed-schema
   /solrconfig.xml
 /configset2
  /conf
   /managed-schema
   /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, if you do this via the core admin API:

```
http://<solr>/admin/cores?action=CREATE&name=mycore&instanceDir=path/to/instance&configSet=configset2
```

Configuration APIs

Solr includes several APIs that can be used to modify settings in `solrconfig.xml`.

- Blob Store API
- Config API
- Request Parameters API
- Managed Resources

Blob Store API

The Blob Store REST API provides REST methods to store, retrieve or list files in a Lucene index. This can be used to upload a jar file which contains standard solr components such as RequestHandlers, SearchComponents, or other custom code you have written for Solr.

When using the blob store, note that the API does not delete or overwrite a previous object if a new one is uploaded with the same name. It always adds a new version of the blob to the index. Deletes can be performed with standard REST delete commands.

The blob store is only available when running in SolrCloud mode. Solr in standalone mode does not support use of a blob store.

The blob store API is implemented as a requestHandler. A special collection named ".system" must be created as the collection that contains the blob store index.

Create a .system Collection

Before using the blob store, a special collection must be created and it must be named `.system`.

The BlobHandler is automatically registered in the .system collection. The `solrconfig.xml`, Schema, and other configuration files for the collection are automatically provided by the system and don't need to be defined specifically.

If you do not use the `-shards` or `-replicationFactor` options, then defaults of 1 shard and 1 replica will be used.

You can create the `.system` collection with the Collections API, as in this example:

```
curl "http://localhost:8983/solr/admin/collections?action=CREATE&name=.system&replicationFactor=2"
```

⚠ Note that the `bin/solr` script cannot be used to create the `.system` collection at this time. Also, please ensure that there is at least one collection created before creating the `.system` collection.

Upload Files to Blob Store

After the `.system` collection has been created, files can be uploaded to the blob store with a request similar to the following:

```
curl -X POST -H 'Content-Type: application/octet-stream' --data-binary @{filename}
http://localhost:8983/solr/.system/blob/{blobname}
```

For example, to upload a file named "test1.jar" as a blob named "test", you would make a POST request like:

```
curl -X POST -H 'Content-Type: application/octet-stream' --data-binary @test1.jar
http://localhost:8983/solr/.system/blob/test
```

A GET request will return the list of blobs and other details:

```
curl http://localhost:8983/solr/.system/blob?omitHeader=true
```

Output:

```
{
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"test/1",
        "md5":"20ff915fa3f5a5d66216081ae705c41b",
        "blobName":"test",
        "version":1,
        "timestamp":"2015-02-04T16:45:48.374Z",
        "size":13108}]
  }
}
```

Details on individual blobs can be accessed with a request similar to:

```
curl http://localhost:8983/solr/.system/blob/{blobname}
```

For example, this request will return only the blob named 'test':

```
curl http://localhost:8983/solr/.system/blob/test?omitHeader=true
```

Output:

```
{
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"test/1",
        "md5":"20ff915fa3f5a5d66216081ae705c41b",
        "blobName":"test",
        "version":1,
        "timestamp":"2015-02-04T16:45:48.374Z",
        "size":13108}]
  }
}
```

The filestream response writer can return a particular version of a blob for download, as in:

```
curl http://localhost:8983/solr/.system/blob/{blobname}/{version}?wt=filestream >
{outputfilename}
```

For the latest version of a blob, the {version} can be omitted,

```
curl http://localhost:8983/solr/.system/blob/{blobname}?wt=filestream >
{outputfilename}
```

Use a Blob in a Handler or Component

To use the blob as the class for a request handler or search component, you create a request handler in `solrconfig.xml` as usual. You will need to define the following parameters:

- `class`: the fully qualified class name. For example, if you created a new request handler class called CRUDHandler, you would enter `org.apache.solr.core.CRUDHandler`.
- `runtimeLib`: Set to true to require that this component should be loaded from the classloader that loads the runtime jars.

For example, to use a blob named test, you would configure `solrconfig.xml` like this:

```
<requestHandler name="/myhandler" class="org.apache.solr.core.myHandler"
runtimeLib="true" version="1">
</requestHandler>
```

If there are parameters available in the custom handler, you can define them in the same way as any other request handler definition.

Config API

The Config API enables manipulating various aspects of your `solrconfig.xml` using REST-like API calls. This feature is enabled by default and works similarly in both SolrCloud and standalone mode. Many commonly edited properties (such as cache sizes and commit settings) and request handler definitions can be changed with this API.

When using this API, `solrconfig.xml` is is not changed. Instead, all edited configuration is stored in a file called `configoverlay.json`. The values in `configoverlay.json` override the values in `solrconfig.xml`.

- API Entry Points
- Commands
  - Commands for Common Properties
  - Commands for Custom Handlers and Local Components
  - Commands for User-Defined Properties
- How to Map solrconfig.xml Properties to JSON
- Examples
  - Creating and Updating Common Properties
  - Creating and Updating Request Handlers
  - Creating and Updating User-Defined Properties
- How It Works
  - Empty Command
  - Listening to config Changes

API Entry Points

- `/config`: retrieve or modify the config. GET to retrieve and POST for executing commands
- `/config/overlay`: retrieve the details in the `configoverlay.json` alone
- `/config/params` : allows creating parameter sets that can override or take the place of parameters defined in `solrconfig.xml`. See the Request Parameters API section for more details.

Commands

This API uses specific commands to tell Solr what property or type of property to add to `configoverlay.json`. The commands are passed as part of the data sent with the request.

The config commands are categorized into 3 different sections which manipulate various data structures in `solr config.xml`. Each of these is described below.

- Common Properties
- Components
- User-defined properties

Commands for Common Properties

The common properties are those that are frequently need to be customized in a Solr instance. They are manipulated with two commands:

- `set-property`: Set a well known property. The names of the properties are predefined and fixed. If the property has already been set, this command will overwrite the previous setting.
- `unset-property`: Remove a property set using the `set-property` command.

The properties that are configured with these commands are predefined and listed below. The names of these properties are derived from their XML paths as found in `solrconfig.xml`.

- `updateHandler.autoCommit.maxDocs`
- `updateHandler.autoCommit.maxTime`
- `updateHandler.autoCommit.openSearcher`
- `updateHandler.autoSoftCommit.maxDocs`
- `updateHandler.autoSoftCommit.maxTime`
- `updateHandler.commitWithin.softCommit`
- `updateHandler.indexWriter.closeWaitsForMerges`
- `query.filterCache.class`
- `query.filterCache.size`
- `query.filterCache.initialSize`
- `query.filterCache.autowarmCount`
- `query.filterCache.regenerator`
- `query.queryResultCache.class`
- `query.queryResultCache.size`
- `query.queryResultCache.initialSize`
- `query.queryResultCache.autowarmCount`
- `query.queryResultCache.regenerator`
- `query.documentCache.class`
- `query.documentCache.size`
- `query.documentCache.initialSize`
- `query.documentCache.autowarmCount`
- `query.documentCache.regenerator`
- `query.fieldValueCache.class`
- `query.fieldValueCache.size`
- `query.fieldValueCache.initialSize`
- `query.fieldValueCache.autowarmCount`
- `query.fieldValueCache.regenerator`
- `query.useFilterForSortedQuery`

- `query.queryResultWindowSize`
- `query.queryResultMaxDocCached`
- `query.enableLazyFieldLoading`
- `query.boolToFilterOptimizer`
- `query.maxBooleanClauses`
- `jmx.agentId`
- `jmx.serviceUrl`
- `jmx.rootName`
- `requestDispatcher.handleSelect`
- `requestDispatcher.requestParsers.multipartUploadLimitInKB`
- `requestDispatcher.requestParsers.formdataUploadLimitInKB`
- `requestDispatcher.requestParsers.enableRemoteStreaming`
- `requestDispatcher.requestParsers.addHttpRequestToContext`

Commands for Custom Handlers and Local Components

Custom request handlers, search components, and other types of localized Solr components (such as custom query parsers, update processors, etc.) can be added, updated and deleted with specific commands for the component being modified.

The syntax is similar in each case: `add-<component-name>`, `update-<component-name>`, and `delete-<component-name>`. Please note that the command name is not case sensitive, so `Add-RequestHandler`, `ADD-REQUESTHANDLER` and `add-requesthandler` are all equivalent. In each case, `add-`commands add the new configuration to `configoverlay.json`, which will override any other settings for the component in `solrconfig.xml`; `update-` commands overwrite an existing setting in `configoverlay.json`; and `delete-` commands remove the setting from `configoverlay.json`. Settings removed from `configoverlay.json` are not removed from `solrconfig.xml`.

The full list of available commands follows below:

General Purpose Commands

These commands are the most commonly used:

- `add-requesthandler`
- `update-requesthandler`
- `delete-requesthandler`
- `add-searchcomponent`
- `update-searchcomponent`
- `delete-searchcomponent`
- `add-initparams`
- `update-initparams`
- `delete-initparams`
- `add-queryresponsewriter`
- `update-queryresponsewriter`
- `delete-queryresponsewriter`

Advanced Commands

These commands allow registering more advanced customizations to Solr:

- `add-queryparser`
- `update-queryparser`
- `delete-queryparser`
- `add-valuesourceparser`
- `update-valuesourceparser`
- `delete-valuesourceparser`
- `add-transformer`
- `update-transformer`
- `delete-transformer`

- `add-updateprocessor`
- `update-updateprocessor`
- `delete-updateprocessor`
- `add-queryconverter`
- `update-queryconverter`
- `delete-queryconverter`
- `add-listener`
- `update-listener`
- `delete-listener`
- `add-runtimelib`
- `update-runtimelib`
- `delete-runtimelib`

See the section Creating and Updating Request Handlers below for examples of using these commands.

What about `<updateRequestProcessorChain>`?

The Config API does not let you create or edit `<updateRequestProcessorChain>` elements. However, it is possible to create `<updateProcessor>` entries and can use them by name to create a chain.

example:

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json' -d '{
"add-updateprocessor" : { "name" : "firstFld",
                          "class": "solr.FirstFieldValueUpdateProcessorFactory",
                          "fieldName":"test_s"}}'
```

You can use this directly in your request by adding a parameter in the `<updateRequestProcessorChain>` for the specific update processor called `processor=firstFld`.

Commands for User-Defined Properties

Solr lets users templatize the `solrconfig.xml` using the place holder format `${variable_name:default_val}`. You could set the values using system properties, for example, `-Dvariable_name= my_customvalue`. The same can be achieved during runtime using these commands:

- `set-user-property`: Set a user-defined property. If the property has already been set, this command will overwrite the previous setting.
- `unset-user-property`: Remove a user-defined property.

The structure of the request is similar to the structure of requests using other commands, in the format of `"command":{"variable_name":"property_value"}`. You can add more than one variable at a time if necessary.

For more information about user-defined properties, see the section User defined properties from core.properties . See also the section Creating and Updating User-Defined Properties below for examples of how to use this type of command.

How to Map `solrconfig.xml` Properties to JSON

By using this API, you will be generating JSON representations of properties defined in `solrconfig.xml`. To understand how properties should be represented with the API, let's take a look at a few examples.

Here is what a request handler looks like in `solrconfig.xml`:

```
<requestHandler name="/query" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <str name="wt">json</str>
    <str name="indent">true</str>
  </lst>
</requestHandler>
```

The same request handler defined with the Config API would look like this:

```
{
  "add-requesthandler":{
    "name":"/query",
    "class":"solr.SearchHandler",
    "defaults":{
      "echoParams":"explicit",
      "wt":"json",
      "indent":true
    }
  }
}
```

The QueryElevationComponent searchComponent in `solrconfig.xml` looks like this:

```
<searchComponent name="elevator" class="solr.QueryElevationComponent" >
  <str name="queryFieldType">string</str>
  <str name="config-file">elevate.xml</str>
</searchComponent>
```

And the same searchComponent with the Config API:

```
{
  "add-searchcomponent":{
    "name":"elevator",
    "class":"QueryElevationComponent",
    "queryFieldType":"string",
    "config-file":"elevate.xml"
  }
}
```

Removing the searchComponent with the Config API:

```
{
  "delete-searchcomponent":"elevator"
}
```

A simple highlighter looks like this in `solrconfig.xml` (example has been truncated for space):

```
<searchComponent class="solr.HighlightComponent" name="highlight">
    <highlighting>
      <fragmenter name="gap"
                  default="true"
                  class="solr.highlight.GapFragmenter">
        <lst name="defaults">
          <int name="hl.fragsize">100</int>
        </lst>
      </fragmenter>

      <formatter name="html"
                 default="true"
                 class="solr.highlight.HtmlFormatter">
        <lst name="defaults">
          <str name="hl.simple.pre"><![CDATA[<em>]]></str>
          <str name="hl.simple.post"><![CDATA[</em>]]></str>
        </lst>
      </formatter>

      <encoder name="html" class="solr.highlight.HtmlEncoder" />
...
    </highlighting>
```

The same highlighter with the Config API:

```
{
 "add-searchcomponent": {
  "highlight": {
   "name": "myHighlight",
   "class": "solr.HighlightComponent",
   "": {
    "gap": {
     "default": "true",
     "name": "gap",
     "class": "solr.highlight.GapFragmenter",
     "defaults": {
      "hl.fragsize": 100
     }
    }
   },
   "html": [{
    "default": "true",
    "name": "html",
    "class": "solr.highlight.HtmlFormatter",
    "defaults": {
     "hl.simple.pre": "<![CDATA[<em>]]>",
     "hl.simple.post": "<![CDATA[</em>]]>"
    }
   }, {
    "name": "html",
    "class": "solr.highlight.HtmlEncoder"
   }]
  }
 }
}
```

Set autoCommit properties in `solrconfig.xml`:

```
<autoCommit>
  <maxTime>15000</maxTime>
  <openSearcher>false</openSearcher>
</autoCommit>
```

Define the same properties with the Config API:

```
{
  "set-property": {
    "updateHandler.autoCommit.maxTime":15000,
    "updateHandler.autoCommit.openSearcher":false
  }
}
```

Name Components for the Config API

The Config API always allows changing the configuration of any component by name. However, some configurations such as `listener` or `initParams` do not require a name in `solrconfig.xml`. In order to be able to `update` and `delete` of the same item in `configoverlay.json`, the name attribute becomes mandatory.

Examples

Creating and Updating Common Properties

This change sets the `query.filterCache.autowarmCount`to 1000 items and unsets the `query.filterCache.size`.

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json' -d'{
    "set-property" : {"query.filterCache.autowarmCount":1000},
    "unset-property" :"query.filterCache.size"}'
```

Using the `/config/overlay` endpoint, you can verify the changes with a request like this:

```
curl http://localhost:8983/solr/gettingstarted/config/overlay?omitHeader=true
```

And you should get a response like this:

```
{
  "overlay":{
    "znodeVersion":1,
    "props":{"query":{"filterCache":{
          "autowarmCount":1000,
          "size":25}}}}}
```

Creating and Updating Request Handlers

To create a request handler, we can use the `add-requesthandler` command:

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json'  -d '{
  "add-requesthandler" : {
    "name": "/mypath",
    "class":"solr.DumpRequestHandler",
    "defaults":{ "x":"y" ,"a":"b", "wt":"json", "indent":true },
    "useParams":"x"
  },
}'
```

Make a call to the new request handler to check if it is registered:

```
curl http://localhost:8983/solr/techproducts/mypath?omitHeader=true
```

And you should see the following as output:

```
{
  "params":{
    "indent":"true",
    "a":"b",
    "x":"y",
    "wt":"json"},
  "context":{
    "webapp":"/solr",
    "path":"/mypath",
    "httpMethod":"GET"}}
```

To update a request handler, you should use the `update-requesthandler` command :

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json'  -d '{
  "update-requesthandler": {
    "name": "/mypath",
    "class":"solr.DumpRequestHandler",
    "defaults": { "x":"new value for X", "wt":"json", "indent":true },
    "useParams":"x"
  }
}'
```

As another example, we'll create another request handler, this time adding the 'terms' component as part of the definition:

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json' -d '{
  "add-requesthandler": {
    "name": "/myterms",
    "class":"solr.SearchHandler",
    "defaults": { "terms":true, "distrib":false },
    "components": [ "terms" ]
  }
}'
```

Creating and Updating User-Defined Properties

This command sets a user property.

```
curl http://localhost:8983/solr/techproducts/config
-H'Content-type:application/json' -d '{
    "set-user-property" : {"variable_name":"some_value"}}'
```

Again, we can use the `/config/overlay` endpoint to verify the changes have been made:

```
curl http://localhost:8983/solr/techproducts/config/overlay?omitHeader=true
```

And we would expect to see output like this:

```
{"overlay":{
   "znodeVersion":5,
   "userProps":{
     "variable_name":"some_value"}}
}
```

To unset the variable, issue a command like this:

```
curl http://localhost:8983/solr/techproducts/config
-H'Content-type:application/json' -d '{"unset-user-property" : "variable_name"}'
```

How It Works

Every core watches the ZooKeeper directory for the configset being used with that core. In standalone mode, however, there is no watch (because ZooKeeper is not running). If there are multiple cores in the same node using the same configset, only one ZooKeeper watch is used. For instance, if the configset 'myconf' is used by a core, the node would watch `/configs/myconf`. Every write operation performed through the API would 'touch' the directory (sets an empty byte[] to trigger watches) and all watchers are notified. Every core would check if the Schema file, `solrconfig.xml` or `configoverlay.json` is modified by comparing the `znode` versions and if modified, the core is reloaded.

If `params.json` is modified, the params object is just updated without a core reload (see the section Request Parameters API for more information about `params.json`).

Empty Command

If an empty command is sent to the `/config` endpoint, the watch is triggered on all cores using this configset. For example:

```
curl http://localhost:8983/solr/techproducts/config
-H'Content-type:application/json' -d '{}'
```

Directly editing any files without 'touching' the directory **will not** make it visible to all nodes.

It is possible for components to watch for the configset 'touch' events by registering a listener using `SolrCore#registerConfListener()`.

Listening to config Changes

Any component can register a listener using:

`SolrCore#addConfListener(Runnable listener)`

to get notified for config changes. This is not very useful if the files modified result in core reloads (i.e., `configoverlay.xml` or Schema). Components can use this to reload the files they are interested in.

Request Parameters API

The Request Parameters API allows creating parameter sets that can override or take the place of parameters defined in `solrconfig.xml`. The parameter sets defined with this API can be used in requests to Solr, or referenced directly in `solrconfig.xml` request handler definitions.

It is really another endpoint of the Config API instead of a separate API, and has distinct commands. It does not replace or modify any sections of `solrconfig.xml`, but instead provides another approach to handling parameters used in requests. It behaves in the same way as the Config API, by storing parameters in another file that will be used at runtime. In this case, the parameters are stored in a file named `params.json`. This file is kept in ZooKeeper or in the `conf` directory of a standalone Solr instance.

The settings stored in `params.json` are used at query time to override settings defined in `solrconfig.xml` in some cases as described below.

When might you want to use this feature?

- To avoid frequently editing your `solrconfig.xml` to update request parameters that change often.
- To reuse parameters across various request handlers.
- To mix and match parameter sets at request time.
- To avoid a reload of your collection for small parameter changes.

The Request Parameters Endpoint

All requests are sent to the `/config/params` endpoint of the Config API.

Setting Request Parameters

The request to set, unset, or update request parameters is sent as a set of Maps with names. These objects can be directly used in a request or a request handler definition.

The available commands are:

- `set`: Create or overwrite a parameter set map.
- `unset`: delete a parameter set map.
- `update`: update a parameter set map. This is equivalent to a map.putAll(newMap) . Both the maps are merged and if the new map has same keys as old they are overwritten

You can mix these commands into a single request if necessary.

Each map must include a name so it can be referenced later, either in a direct request to Solr or in a request handler definition.

In the following example, we are setting 2 sets of parameters named 'myFacets' and 'myQueries'.

```
curl http://localhost:8983/solr/techproducts/config/params -H
'Content-type:application/json'  -d '{
  "set":{
    "myFacets":{
      "facet":"true",
      "facet.limit":5}},
  "set":{
    "myQueries":{
      "defType":"edismax",
      "rows":"5",
      "df":"text_all"}}
}'
```

In the above example all the parameters are equivalent to the "defaults" in solrconfig.xml. It is possible to add invariants and appends as follows

```
curl http://localhost:8983/solr/techproducts/config/params -H
'Content-type:application/json'  -d '{
  "set":{
    "my_handler_params":{
      "facet.limit":5,
      "_invariants_": {
        "facet":true,
        "wt":"json"
       },
      "_appends_":{"facet.field":["field1","field2"]
     }
   }}
}'
```

now it is possible to define a request handler as follows

```
<requestHandler name="/my_handler" class="solr.SearchHandler"
useParams="my_handler_params"/>
```

It will be equivalent to a requesthandler definition as follows,

```
<requestHandler name="/my_handler" class="solr.SearchHandler">
<lst name="defaults">
 <int name="facet.limit">5</int>
</lst>
<lst name="invariants>
  <str name="wt">json</>
  <bool name="facet">true<bool>
</lst>
<lst name="appends">
 <arr name="facet.field">
  <str>field1</str>
  <str>field2</str>
</arr>
</lst>
</requestHandler>
```

Update example,

```
curl http://localhost:8983/solr/techproducts/config/params -H
'Content-type:application/json'  -d '{
  "update":{
    "myFacets":{
      "facet.limit":10}},
}'
```

This command will add (or replace) the `facet.limit` param to the `myFacets` map, keeping all other existing `myFacets` params.

To see the parameters that have been set, you can use the `/config/params` endpoint to read the contents of `params.json`, or use the name in the request:

```
curl http://localhost:8983/solr/techproducts/config/params

#Or use the params name
curl http://localhost:8983/solr/techproducts/config/params/myQueries
```

The `useParams` Parameter

When making a request, the `useParams` parameter applies the request parameters set to the request. This is translated at request time to the actual params.

For example (using the names we set up in the earlier example, please replace with your own name):

```
http://localhost/solr/techproducts/select?useParams=myQueries
```

It is possible to pass more than one parameter set in the same request. For example:

```
http://localhost/solr/techproducts/select?useParams=myFacets,myQueries
```

In the above example the param set 'myQueries' is applied on top of 'myFacets'. So, values in 'myQueries' take precedence over values in 'myFacets'. Additionally, any values passed in the request take precedence over 'useParams' params. This acts like the "defaults" specified in the '`<requestHandler>`' definition in `solrconfig.xml`.

The parameter sets can be used directly in a request handler definition as follows. Please note that the 'useParams' specified is always applied even if the request contains `useParams`.

```
<requestHandler name="/terms" class="solr.SearchHandler" useParams="myQueries">
  <lst name="defaults">
    <bool name="terms">true</bool>
    <bool name="distrib">false</bool>
  </lst>
  <arr name="components">
    <str>terms</str>
  </arr>
</requestHandler>
```

To summarize, parameters are applied in this order:

- parameters defined in `<invariants>` in `solrconfig.xml`.
- parameters applied in _invariants_ in params.json and that is specified in the requesthandler definition or even in request
- parameters defined in the request directly.
- parameter sets defined in the request, in the order they have been listed with `useParams`.
- parameter sets defined in `params.json` that have been defined in the request handler.
- parameters defined in `<defaults>` in `solrconfig.xml`.

Public APIs

The RequestParams Object can be accessed using the method `SolrConfig#getRequestParams()`. Each paramset can be accessed by their name using the method `RequestParams#getRequestParams(String name)`.

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.

All of the examples in this section assume you are running the "techproducts" Solr example:

```
bin/solr -e techproducts
```

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 techproducts collection is: `/solr/techproducts/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/techproducts/schema/analysis/stopwords/french`.

So now let's see this API in action, starting with a simple GET request:

```
curl "http://localhost:8983/solr/techproducts/schema/analysis/stopwords/english"
```

Assuming you sent this request to Solr, 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 `sample_techproducts_configs` config set ships with a pre-built set of managed stop words, 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/techproducts/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/techproducts/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/techproducts/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 `sample_techproducts_configs` config set includes a pre-built set of synonym mappings suitable for the sample data 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 synonyms, send a GET request to:

```
curl "http://localhost:8983/solr/techproducts/schema/analysis/synonyms/english"
```

This request will return a response that looks like:

```
{
  "responseHeader":{
    "status":0,
    "QTime":3},
  "synonymMappings":{
    "initArgs":{
      "ignoreCase":true,
      "format":"solr"},
    "initializedOn":"2014-12-16T22:44:05.33Z",
    "managedMap":{
      "GB":
        ["GiB",
         "Gigabyte"],
      "TV":
        ["Television"],
      "happy":
        ["glad",
         "joyful"]}}}
```

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/techproducts/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"]`.

You can also PUT a list of symmetric synonyms, which will be expanded into a mapping for each term in the list. For example, you could PUT the following list of symmetric synonyms using the JSON list syntax instead of a map:

```
curl -X PUT -H 'Content-type:application/json' --data-binary '["funny",
"entertaining", "whimiscal", "jocular"]'
"http://localhost:8983/solr/techproducts/schema/analysis/synonyms/english"
```

Note that the expansion is performed when processing the PUT request so the underlying persistent state is still a managed map. Consequently, if after sending the previous PUT request, you did a GET for `/schema/analysis/synonyms/english/jocular`, then you would receive a list containing `["funny", "entertaining", "whimiscal"]`. Once you've created synonym mappings using a list, each term must be managed separately.

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 for each collection. 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/techproducts/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$SynonymMan
ager",
      "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/techproducts/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/techproducts/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/techproducts/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/techproducts/schema/analysis/stopwords/german"
```

Related Topics

- Using Solr's REST APIs to manage stop words and synonyms by Tim Potter @ SearchHub.org

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.

If you have a plugin you would like to use, and you are running in SolrCloud mode, you can use the Blob Store API and the Config API to load the jars to Solr. The commands to use are described in the section Adding Custom Plugins in SolrCloud Mode.

Adding Custom Plugins in SolrCloud Mode

When running Solr in SolrCloud mode and you want to use custom code (such as custom analyzers, tokenizers, query parsers, and other plugins), it can be cumbersome to add jars to the classpath on all nodes in your cluster. Using the Blob Store API and special commands with the Config API, you can upload jars to a special system-level collection and dynamically load plugins from them at runtime with out needing to restart any nodes.



This Feature is Disabled By Default

In addition to requiring that Solr by running in SolrCloud mode, this feature is also disabled by default unless all Solr nodes are run with the `-Denable.runtime.lib=true` option on startup.

Before enabling this feature, users should carefully consider the issues discussed in the Securing Runtime Libraries section below.

Uploading Jar Files

The first step is to use the Blob Store API to upload your jar files. This will to put your jars in the `.system` collecti on and distribute them across your SolrCloud nodes. These jars are added to a separate classloader and only accessible to components that are configured with the property `runtimeLib=true`. These components are loaded lazily because the `.system` collection may not be loaded when a particular core is loaded.

Config API Commands to use Jars as Runtime Libraries

The runtime library feature uses a special set of commands for the Config API to add, update, or remove jar files currently available in the blob store to the list of runtime libraries.

The following commands are used to manage runtime libs:

- `add-runtimelib`
- `update-runtimelib`
- `delete-runtimelib`

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json' -d '{
   "add-runtimelib": { "name":"jarblobname", "version":2 },
   "update-runtimelib": { "name":"jarblobname", "version":3 },
   "delete-runtimelib": "jarblobname"
}'
```

The name to use is the name of the blob that you specified when you uploaded your jar to the blob store. You should also include the version of the jar found in the blob store that you want to use. These details are added to `configoverlay.json`.

The default `SolrResourceLoader` does not have visibility to the jars that have been defined as runtime libraries. There is a classloader that can access these jars which is made available only to those components which are specially annotated.

Every pluggable component can have an optional extra attribute called `runtimeLib=true`, which means that the components are not loaded at core load time. Instead, they will be loaded on demand. If all the dependent jars are not available when the component is loaded, an error is thrown.

This example shows creating a ValueSourceParser using a jar that has been loaded to the Blob store.

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json' -d '{
  "create-valuesourceparser": {
    "name": "nvl",
    "runtimeLib": true,
    "class": "solr.org.apache.solr.search.function.NvlValueSourceParser,
    "nvlFloatValue": 0.0 }
}'
```

Securing Runtime Libraries

A drawback of this feature is that it could be used to load malicious executable code into the system. However, it is possible to restrict the system to load only trusted jars using PKI to verify that the executables loaded into the system are trustworthy.

The following steps will allow you enable security for this feature. The instructions assume you have started all your Solr nodes with the `-Denable.runtime.lib=true`.

Step 1: Generate an RSA Private Key

The first step is to generate an RSA private key. The example below uses a 512-bit key, but you should use the strength appropriate to your needs.

```
$ openssl genrsa -out priv_key.pem 512
```

Step 2: Output the Public Key

The public portion of the key should be output in DER format so Java can read it.

```
$ openssl rsa -in priv_key.pem -pubout -outform DER -out pub_key.der
```

Step 3: Load the Key to ZooKeeper

The .der files that are output from Step 2 should then be loaded to ZooKeeper under a node `/keys/exe` so they are available throughout every node. You can load any number of public keys to that node and all are valid. If a key is removed from the directory, the signatures of that key will cease to be valid. So, before removing the a key, make sure to update your runtime library configurations with valid signatures with the `update-runtimelib` command.

At the current time, you can only use the ZooKeeper `zkCli.sh` (or `zkCli.cmd` on Windows) script to issue these commands (the Solr version has the same name, but is not the same). If you are running the embedded ZooKeeper that is included with Solr, you **do not** have this script already; in order to use it, you will need to download a copy of ZooKeeper v3.4.6 from http://zookeeper.apache.org/. Don't worry about configuring the download, you're just trying to get the command line utility script. When you start the script, you will connect to the embedded ZooKeeper. If you have your own ZooKeeper ensemble running already, you can find the script in `$ZK_INSTALL/bin/zkCli.sh` (or `zkCli.cmd` if you are using Windows).

To load the keys, you will need to connect to ZooKeeper with `zkCli.sh`, create the directories, and then create the key file, as in the following example.

```
# Connect to ZooKeeper
# Replace the server location below with the correct ZooKeeper connect string for
your installation.
$ .bin/zkCli.sh -server localhost:9983

# After connection, you will interact with the ZK prompt.
# Create the directories
[zk: localhost:9983(CONNECTED) 5] create /keys
[zk: localhost:9983(CONNECTED) 5] create /keys/exe

# Now create the public key file in ZooKeeper
# The second path is the path to the .der file on your local machine
[zk: localhost:9983(CONNECTED) 5] create /keys/exe/pub_key.der
/myLocal/pathTo/pub_key.der
```

After this, any attempt to load a jar will fail. All your jars must be signed with one of your private keys for Solr to trust it. The process to sign your jars and use the signature is outlined in Steps 4-6.

Step 4: Sign the jar File

Next you need to sign the sha1 digest of your jar file and get the base64 string.

```
$ openssl dgst -sha1 -sign priv_key.pem myjar.jar | openssl enc -base64
```

The output of this step will be a string that you will need to add the jar to your classpath in Step 6 below.

Step 5: Load the jar to the Blob Store

Load your jar to the Blob store, using the Blob Store API. This step does not require a signature; you will need the signature in Step 6 to add it to your classpath.

```
curl -X POST -H 'Content-Type: application/octet-stream' --data-binary @{filename}
http://localhost:8983/solr/.system/blob/{blobname}
```

The blob name that you give the jar file in this step will be used as the name in the next step.

Step 6: Add the jar to the Classpath

Finally, add the jar to the classpath using the Config API as detailed above. In this step, you will need to provide the signature of the jar that you got in Step 4.

```
curl http://localhost:8983/solr/techproducts/config -H
'Content-type:application/json'  -d '{
  "add-runtimelib": {
    "name":"blobname",
    "version":2,

"sig":"mW1Gwtz2QazjfVdrLFHfbGwcr8xzFYgUOLu68LHqWRDvLG0uLcy1McQ+AzVmeZFBf1yLPDEHBWJb5
KXr8bdbHN/

PYgUB1nsr9pk4EFyD9KfJ8TqeH/ijQ9waa/vjqyiKEI9U550EtSzruLVZ32wJ7smvV0fj2YYhrUaaPzOn9g0
=" }
}'
```

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. 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:

Taking Solr to Production: Describes how to install Solr as a service on Linux for production environments.

Securing Solr: How to use the Basic and Kerberos authentication and rule-based authorization plugins for Solr, and how to enable SSL.

Running Solr on HDFS: How to use HDFS to store your Solr indexes and transaction logs.

Making and Restoring Backups: Describes backup strategies for your Solr indexes.

Configuring Logging: Describes how to configure logging for Solr.

Using JMX with Solr: Describes how to use Java Management Extensions with Solr.

MBean Request Handler: How to use Solr's MBeans for programmatic access to the system plugins and stats.

Taking Solr to Production

This section provides guidance on how to setup Solr to run in production on *nix platforms, such as Ubuntu. Specifically, we'll walk through the process of setting up to run a single Solr instance on a Linux host and then provide tips on how to support multiple Solr nodes running on the same host.

- Service Installation Script
  - Planning your directory structure
    - Solr Installation Directory
    - Separate Directory for Writable Files
  - Create the Solr user
  - Run the Solr Installation Script
    - Solr Home Directory
    - Environment overrides include file
    - Log settings
    - init.d script
  - Progress Check
- Fine tune your production setup
  - Memory and GC Settings
    - Out-of-Memory Shutdown Hook
  - SolrCloud
    - ZooKeeper chroot
  - Solr Hostname
  - Override settings in solrconfig.xml
  - Enable Remote JMX Access
- Running multiple Solr nodes per host

Service Installation Script

Solr includes a service installation script (`bin/install_solr_service.sh`) to help you install Solr as a service on Linux. Currently, the script only supports Red Hat, Ubuntu, Debian, and SUSE Linux distributions. Before running the script, you need to determine a few parameters about your setup. Specifically, you need to decide where to install Solr and which system user should be the owner of the Solr files and process.

Planning your directory structure

We recommend separating your live Solr files, such as logs and index files, from the files included in the Solr distribution bundle, as that makes it easier to upgrade Solr and is considered a good practice to follow as a system administrator.

Solr Installation Directory

By default, the service installation script will extract the distribution archive into `/opt`. You can change this location using the `-i` option when running the installation script. The script will also create a symbolic link to the versioned directory of Solr. For instance, if you run the installation script for Solr X.0.0, then the following directory structure will be used:

```
/opt/solr-X.0.0
/opt/solr -> /opt/solr-X.0.0
```

Using a symbolic link insulates any scripts from being dependent on the specific Solr version. If, down the road, you need to upgrade to a later version of Solr, you can just update the symbolic link to point to the upgraded version of Solr. We'll use `/opt/solr` to refer to the Solr installation directory in the remaining sections of this page.

Separate Directory for Writable Files

You should also separate writable Solr files into a different directory; by default, the installation script uses `/var/solr`, but you can override this location using the `-d` option. With this approach, the files in `/opt/solr` will remain untouched and all files that change while Solr is running will live under `/var/solr`.

Create the Solr user

Running Solr as `root` is not recommended for security reasons. Consequently, you should determine the username of a system user that will own all of the Solr files and the running Solr process. By default, the installation script will create the **solr** user, but you can override this setting using the -u option. If your organization has specific requirements for creating new user accounts, then you should create the user before running the script. The installation script will make the Solr user the owner of the `/opt/solr` and `/var/solr` directories.

You are now ready to run the installation script.

Run the Solr Installation Script

To run the script, you'll need to download the latest Solr distribution archive and then do the following (NOTE: replace `solr-X.Y.Z` with the actual version number):

```
$ tar xzf solr-X.Y.Z.tgz solr-X.Y.Z/bin/install_solr_service.sh --strip-components=2
```

The previous command extracts the `install_solr_service.sh` script from the archive into the current directory. If installing on Red Hat, please make sure **lsof** is installed before running the Solr installation script (`sudo yum install lsof`). The installation script must be run as root:

```
$ sudo bash ./install_solr_service.sh solr-X.Y.Z.tgz
```

By default, the script extracts the distribution archive into `/opt`, configures Solr to write files into `/var/solr`, and runs Solr as the `solr` user. Consequently, the following command produces the same result as the previous

command:

```
$ sudo bash ./install_solr_service.sh solr-X.Y.Z.tgz -i /opt -d /var/solr -u solr -s
solr -p 8983
```

You can customize the service name, installation directories, port, and owner using options passed to the installation script. To see available options, simply do:

```
$ sudo bash ./install_solr_service.sh -help
```

Once the script completes, Solr will be installed as a service and running in the background on your server (on port 8983). To verify, you can do:

```
$ sudo service solr status
```

We'll cover some additional configuration settings you can make to fine-tune your Solr setup in a moment. Before moving on, let's take a closer look at the steps performed by the installation script. This gives you a better overview and will help you understand important details about your Solr installation when reading other pages in this guide; such as when a page refers to Solr home, you'll know exactly where that is on your system.

Solr Home Directory

The Solr home directory (not to be confused with the Solr installation directory) is where Solr manages core directories with index files. By default, the installation script uses `/var/solr/data`. If the `-d` option is used on the install script, then this will change to the `data` subdirectory in the location given to the -d option. Take a moment to inspect the contents of the Solr home directory on your system. If you do not store `solr.xml` in ZooKeeper, the home directory must contain a `solr.xml` file. When Solr starts up, the Solr start script passes the location of the home directory using the `-Dsolr.solr.home` system property.

Environment overrides include file

The service installation script creates an environment specific include file that overrides defaults used by the `bin/solr` script. The main advantage of using an include file is that it provides a single location where all of your environment-specific overrides are defined. Take a moment to inspect the contents of the `/etc/default/solr.in.sh` file, which is the default path setup by the installation script. If you used the `-s` option on the install script to change the name of the service, then the first part of the filename will be different. For a service named `solr-demo`, the file will be named `/etc/default/solr-demo.in.sh`. There are many settings that you can override using this file. However, at a minimum, this script needs to define the `SOLR_PID_DIR` and `SOLR_HOME` variables, such as:

```
SOLR_PID_DIR=/var/solr
SOLR_HOME=/var/solr/data
```

The `SOLR_PID_DIR` variable sets the directory where the start script will write out a file containing the Solr server's process ID.

Log settings

Solr uses Apache Log4J for logging. The installation script copies `/opt/solr/server/resources/log4j.properties` to `/var/solr/log4j.properties` and customizes it for your environment. Specifically it updates the Log4J settings to create logs in the `/var/solr/logs` directory. Take a moment to verify that the Solr include file is configured to send logs to the correct location by checking the following settings in `/etc/default/solr.in.sh`:

```
LOG4J_PROPS=/var/solr/log4j.properties
SOLR_LOGS_DIR=/var/solr/logs
```

For more information about Log4J configuration, please see: Configuring Logging

init.d script

When running a service like Solr on Linux, it's common to setup an init.d script so that system administrators can control Solr using the service tool, such as: `service solr start`. The installation script creates a very basic init.d script to help you get started. Take a moment to inspect the `/etc/init.d/solr` file, which is the default script name setup by the installation script. If you used the `-s` option on the install script to change the name of the service, then the filename will be different. Notice that the following variables are setup for your environment based on the parameters passed to the installation script:

```
SOLR_INSTALL_DIR=/opt/solr
SOLR_ENV=/etc/default/solr.in.sh
RUNAS=solr
```

The `SOLR_INSTALL_DIR` and `SOLR_ENV` variables should be self-explanatory. The `RUNAS` variable sets the owner of the Solr process, such as `solr`; if you don't set this value, the script will run Solr as **root**, which is not recommended for production. You can use the `/etc/init.d/solr` script to start Solr by doing the following as root:

```
# service solr start
```

The `/etc/init.d/solr` script also supports the **stop**, **restart**, and **status** commands. Please keep in mind that the init script that ships with Solr is very basic and is intended to show you how to setup Solr as a service. However, it's also common to use more advanced tools like **supervisord** or **upstart** to control Solr as a service on Linux. While showing how to integrate Solr with tools like supervisord is beyond the scope of this guide, the `init.d/solr` script should provide enough guidance to help you get started. Also, the installation script sets the Solr service to start automatically when the host machine initializes.

Progress Check

In the next section, we cover some additional environment settings to help you fine-tune your production setup. However, before we move on, let's review what we've achieved thus far. Specifically, you should be able to control Solr using `/etc/init.d/solr`. Please verify the following commands work with your setup:

```
$ sudo service solr restart
$ sudo service solr status
```

The status command should give some basic information about the running Solr node that looks similar to:

```
Solr process PID running on port 8983
{
  "version":"5.0.0 - ubuntu - 2014-12-17 19:36:58",
  "startTime":"2014-12-19T19:25:46.853Z",
  "uptime":"0 days, 0 hours, 0 minutes, 8 seconds",
  "memory":"85.4 MB (%17.4) of 490.7 MB"}
```

If the `status` command is not successful, look for error messages in `/var/solr/logs/solr.log`.

Fine tune your production setup

Memory and GC Settings

By default, the `bin/solr` script sets the maximum Java heap size to 512M (-Xmx512m), which is fine for getting started with Solr. For production, you'll want to increase the maximum heap size based on the memory requirements of your search application; values between 10 and 20 gigabytes are not uncommon for production servers. When you need to change the memory settings for your Solr server, use the `SOLR_JAVA_MEM` variable in the include file, such as:

```
SOLR_JAVA_MEM="-Xms10g -Xmx10g"
```

Also, the include file comes with a set of pre-configured Java Garbage Collection settings that have shown to work well with Solr for a number of different workloads. However, these settings may not work well for your specific use of Solr. Consequently, you may need to change the GC settings, which should also be done with the `GC_TUNE` variable in the `/etc/default/solr.in.sh` include file. For more information about tuning your memory and garbage collection settings, see: JVM Settings.

Out-of-Memory Shutdown Hook

The `bin/solr` script registers the `bin/oom_solr.sh` script to be called by the JVM if an OutOfMemoryError occurs. The `oom_solr.sh` script will issue a `kill -9` to the Solr process that experiences the `OutOfMemoryError`. This behavior is recommended when running in SolrCloud mode so that ZooKeeper is immediately notified that a node has experienced a non-recoverable error. Take a moment to inspect the contents of the `/opt/solr/bin/oom_solr.sh` script so that you are familiar with the actions the script will perform if it is invoked by the JVM.

SolrCloud

To run Solr in SolrCloud mode, you need to set the `ZK_HOST` variable in the include file to point to your ZooKeeper ensemble. Running the embedded ZooKeeper is not supported in production environments. For instance, if you have a ZooKeeper ensemble hosted on the following three hosts on the default client port 2181 (zk1, zk2, and zk3), then you would set:

```
ZK_HOST=zk1,zk2,zk3
```

When the `ZK_HOST` variable is set, Solr will launch in "cloud" mode.

ZooKeeper chroot

If you're using a ZooKeeper instance that is shared by other systems, it's recommended to isolate the SolrCloud znode tree using ZooKeeper's chroot support. For instance, to ensure all znodes created by SolrCloud are stored under `/solr`, you can put `/solr` on the end of your `ZK_HOST` connection string, such as:

```
ZK_HOST=zk1,zk2,zk3/solr
```

Before using a chroot for the first time, you need to create the root path (znode) in ZooKeeper by using the `zkcli.sh` script. We can use the makepath command for that:

```
$ server/scripts/cloud-scripts/zkcli.sh -zkhost zk1,zk2,zk3 -cmd makepath /solr
```

If you also want to bootstrap ZooKeeper with existing `solr_home`, you can instead use use `zkcli.sh` / `zkcli.bat`'s `bootstrap` command, which will also create the chroot path if it does not exist. See Command Line Utilities for more info.

Solr Hostname

Use the `SOLR_HOST` variable in the include file to set the hostname of the Solr server.

```
SOLR_HOST=solr1.example.com
```

Setting the hostname of the Solr server is recommended, especially when running in SolrCloud mode, as this determines the address of the node when it registers with ZooKeeper.

Override settings in solrconfig.xml

Solr allows configuration properties to be overridden using Java system properties passed at startup using the `-Dproperty=value` syntax. For instance, in `solrconfig.xml`, the default auto soft commit settings are set to:

```
<autoSoftCommit>
  <maxTime>${solr.autoSoftCommit.maxTime:-1}</maxTime>
</autoSoftCommit>
```

In general, whenever you see a property in a Solr configuration file that uses the `${solr.PROPERTY:DEFAULT_VALUE}` syntax, then you know it can be overridden using a Java system property. For instance, to set the maxTime for soft-commits to be 10 seconds, then you can start Solr with `-Dsolr.autoSoftCommit.maxTime=10000`, such as:

```
$ bin/solr start -Dsolr.autoSoftCommit.maxTime=10000
```

The `bin/solr` script simply passes options starting with `-D` on to the JVM during startup. For running in production, we recommend setting these properties in the `SOLR_OPTS` variable defined in the include file. Keeping with our soft-commit example, in `/etc/default/solr.in.sh`, you would do:

```
SOLR_OPTS="$SOLR_OPTS -Dsolr.autoSoftCommit.maxTime=10000"
```

Enable Remote JMX Access

If you need to attach a JMX-enabled Java profiling tool, such as JConsole or VisualVM, to a remote Solr server, then you need to enable remote JMX access when starting the Solr server. Simply change the `ENABLE_REMOTE_JMX_OPTS` property in the include file to true. You'll also need to choose a port for the JMX RMI connector to bind to, such as 18983. For example, if your Solr include script sets:

```
ENABLE_REMOTE_JMX_OPTS=true
RMI_PORT=18983
```

The JMX RMI connector will allow Java profiling tools to attach to port 18983. When enabled, the following properties are passed to the JVM when starting Solr:

```
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.local.only=false \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.port=18983 \
-Dcom.sun.management.jmxremote.rmi.port=18983
```

We don't recommend enabling remote JMX access in production, but it can sometimes be useful when doing performance and user-acceptance testing prior to going into production.

Running multiple Solr nodes per host

The `bin/solr` script is capable of running multiple instances on one machine, but for a **typical** installation, this is not a recommended setup. Extra CPU and memory resources are required for each additional instance. A single instance is easily capable of handling multiple indexes.

When to ignore the recommendation
For every recommendation, there are exceptions. For the recommendation above, that exception is mostly applicable when discussing extreme scalability. The best reason for running multiple Solr nodes on one host is decreasing the need for extremely large heaps.

When the Java heap gets very large, it can result in extremely long garbage collection pauses, even with the GC tuning that the startup script provides by default. The exact point at which the heap is considered "very large" will vary depending on how Solr is used. This means that there is no hard number that can be given as a threshold, but if your heap is reaching the neighborhood of 16 to 32 gigabytes, it might be time to consider splitting nodes. Ideally this would mean more machines, but budget constraints might make that impossible.

There is another issue once the heap reaches 32GB. Below 32GB, Java is able to use compressed pointers, but above that point, larger pointers are required, which uses more memory and slows down the JVM.

Because of the potential garbage collection issues and the particular issues that happen at 32GB, if a single instance would require a 64GB heap, performance is likely to improve greatly if the machine is set up with two nodes that each have a 31GB heap.

If your use case requires multiple instances, at a minimum you will need unique Solr home directories for each node you want to run; ideally, each home should be on a different physical disk so that multiple Solr nodes don't have to compete with each other when accessing files on disk. Having different Solr home directories implies that you'll need a different include file for each node. Moreover, if using the `/etc/init.d/solr` script to control Solr as a service, then you'll need a separate script for each node. The easiest approach is to use the service installation script to add multiple services on the same host, such as:

```
$ sudo bash ./install_solr_service.sh solr-X.Y.Z.tgz -s solr2 -p 8984
```

The command shown above will add a service named `solr2` running on port 8984 using `/var/solr2` for writable (aka "live") files; the second server will still be owned and run by the `solr` user and will use the Solr distribution files in `/opt`. After installing the solr2 service, verify it works correctly by doing:

```
$ sudo service solr2 restart
$ sudo service solr2 status
```

Securing Solr

When planning how to secure Solr, you should consider which of the available features or approaches are right for you.

- Authentication or authorization of users using:
  - Kerberos Authentication Plugin
  - Basic Authentication Plugin
  - Rule-Based Authorization Plugin
  - Custom authentication or authorization plugin
- Enabling SSL
- If using SolrCloud, ZooKeeper Access Control

No Solr API, including the Admin UI, is designed to be exposed to non-trusted parties. Tune your firewall so that only trusted computers and people are allowed access. Because of this, the project will not regard e.g. Admin UI XSS issues as security vulnerabilities. However, we still ask you to report such issues in JIRA.

Authentication and Authorization Plugins

Solr has security frameworks for supporting authentication and authorization of users. This allows for verifying a user's identity and for restricting access to resources in a Solr cluster. Solr includes plugins to support Basic authentication, Kerberos, and rule-based authorization of users. Additional plugins can be developed using the authentication and authorization frameworks described below.

The plugin implementation will dictate if the plugin can be used with Solr running in SolrCloud mode only or also if running in standalone mode. If the plugin supports SolrCloud only, a `security.json` file must be created and uploaded to ZooKeeper before it can be used. If the plugin also supports standalone mode, a system property `-DauthenticationPlugin=<pluginClassName>` can be used instead of creating and managing `security.json` in ZooKeeper. Here is a list of the available plugins and the approach supported:

- Basic authentication: SolrCloud only.
- Kerberos authentication: SolrCloud or standalone mode.
- Rule-based authorization: SolrCloud only.

The following section describes how to enable plugins with `security.json` in ZooKeeper when using Solr in SolrCloud mode.

Enable Plugins with security.json

All of the information required to initialize either type of security plugin is stored in a `/security.json` file in ZooKeeper. This file contains 2 sections, one each for authentication and authorization.

security.json

```
{
  "authentication" : {
    "class": "class.that.implements.authentication"
  },
  "authorization": {
    "class": "class.that.implements.authorization"
  }
}
```

The `/security.json` file needs to be in ZooKeeper before a Solr instance comes up so Solr starts with the security plugin enabled. See the section Adding security.json to Zookeeper below for information on how to do this.

Depending on the plugin(s) in use, other information will be stored in `security.json` such as user information or rules to create roles and permissions. This information is added through the APIs for each plugin provided by Solr, or, in the case of a custom plugin, the approach designed by you.

Here is a more detailed `security.json` example. In this, the Basic authentication and rule-based authorization plugins are enabled, and some data has been added:

```
{
"authentication":{
   "class":"solr.BasicAuthPlugin",
   "credentials":{"solr":"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0=
Ndd7LKvVBAaZIF0QAVi1ekCfAJXr1GGfLtRUXhgrF8c="}
},
"authorization":{
   "class":"solr.RuleBasedAuthorizationPlugin",
   "permissions":[{"name":"security-edit",
      "role":"admin"}]
   "user-role":{"solr":"admin"}
}}
```

Adding security.json to ZooKeeper

While configuring Solr to use an authentication or authorization plugin, you will need to upload a `security.json` file to ZooKeeper as in the example below.

```
> server/scripts/cloud-scripts/zkcli.sh -zkhost localhost:2181 -cmd put
/security.json
  '{"authentication": {"class": "org.apache.solr.security.KerberosPlugin"}}'
```

Note that this example defines the `KerberosPlugin` for authentication. You will want to modify this section as appropriate for the plugin you are using.

This example also defines `security.json` on the command line, but you can also define a file locally and upload it to ZooKeeper.



Depending on the authentication and authorization plugin that you use, you may have user information stored in `security.json`. If so, we highly recommend that you implement access control in your ZooKeeper nodes. Information about how to enable this is available in the section ZooKeeper Access Control.

Authentication

Authentication plugins help in securing the endpoints of Solr by authenticating incoming requests. A custom plugin can be implemented by extending the AuthenticationPlugin class.

An authentication plugin consists of two parts:

1. Server-side component, which intercepts and authenticates incoming requests to Solr using a mechanism defined in the plugin, such as Kerberos, Basic Auth or others.
2. Client-side component, i.e., an extension of `HttpClientConfigurer`, which enables a SolrJ client to make requests to a secure Solr instance using the authentication mechanism which the server understands.

Enabling a Plugin

- Specify the authentication plugin in `/security.json` as in this example:

security.json

```
{
  "authentication": {
    "class": "class.that.implements.authentication",
    "other_data" : "..."}
}
```

- All of the content in the authentication block of `security.json` would be passed on as a map to the plugin during initialization.
- An authentication plugin can also be used with a standalone Solr instance by passing in `-DauthenticationPlugin=<plugin class name>` during the startup.

Available Authentication Plugins

Solr has two implementations of authentication plugins:

- Kerberos Authentication Plugin
- Basic Authentication Plugin

Authorization

An authorization plugin can be written for Solr by extending the AuthorizationPlugin interface.

Loading a Custom Plugin

- Make sure that the plug-in implementation is in the classpath.
- The plugin can then be initialized by specifying the same in `security.json` in the following manner:

security.json

```
{
  "authorization": {
    "class": "org.apache.solr.security.MockAuthorizationPlugin",
    "other_data" : "..."}
}
```

All of the content in the `authorization` block of `security.json` would be passed on as a map to the plugin

during initialization.



The authorization plugin is only supported in SolrCloud mode. Also, reloading the plugin isn't supported at this point and requires a restart of the Solr instance (meaning, the JVM should be restarted, not simply a core reload).

Available Authorization Plugins

Solr has one implementation of an authorization plugin:

- Rule-Based Authorization Plugin

Basic Authentication Plugin

Solr can support Basic authentication for users with the use of the BasicAuthPlugin.

An authorization plugin is also available to configure Solr with permissions to perform various activities in the system. The authorization plugin is described in the section Rule-Based Authorization Plugin.

Enable Basic Authentication

To use Basic authentication, you must first create a `security.json` file and store it in ZooKeeper. This file and how to upload it to ZooKeeper is described in detail in the section Enable Plugins with security.json.

For Basic authentication, the `security.json` file must have an `authentication` part which defines the class being used for authentication. Usernames and passwords (as a sha256(password+salt) hash) could be added when the file is created, or can be added later with the Basic authentication API, described below.

The `authorization` part is not related to Basic authentication, but is a separate authorization plugin designed to support fine-grained user access control. For more information, see Rule-Based Authorization Plugin.

An example `security.json` showing both sections is shown below to show how these plugins can work together:

```
{
"authentication":{
   "blockUnknown": true,
   "class":"solr.BasicAuthPlugin",
   "credentials":{"solr":"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0=
Ndd7LKvVBAaZIF0QAVi1ekCfAJXr1GGfLtRUXhgrF8c="}
},
"authorization":{
   "class":"solr.RuleBasedAuthorizationPlugin",
   "permissions":[{"name":"security-edit",
      "role":"admin"}],
   "user-role":{"solr":"admin"}
}}
```

Save the above json to a file called security.json locally. Run the following command to upload it to Zookeeper. (ensure that the Zookeeper port is correct)

```
server/scripts/cloud-scripts/zkcli.sh -zkhost localhost:9983 -cmd putfile
/security.json security.json
```

There are several things defined in this file:

- Basic authentication and rule-based authorization plugins are enabled.
- A user called 'solr', with a password `'SolrRocks'` has been defined.
- `'blockUknown:true'` means that unauthenticated requests are not allowed to pass through
- The 'admin' role has been defined, and it has permission to edit security settings.
- The 'solr' user has been defined to the 'admin' role.

Caveats

There are a few things to keep in mind when using the Basic authentication plugin.

- Credentials are sent in plain text by default. It's recommended to use SSL for communication when Basic authentication is enabled, as described in the section Enabling SSL.
- A user who has access to write permissions to `security.json` will be able to modify all the permissions and how users have been assigned permissions. Special care should be taken to only grant access to editing security to appropriate users.
- Your network should, of course, be secure. Even with Basic authentication enabled, you should not unnecessarily expose Solr to the outside world.

Editing Authentication Plugin Configuration

An Authentication API allows modifying user IDs and passwords. The API provides an endpoint with specific commands to set user details or delete a user.

API Entry Point

`admin/authentication`

This endpoint is not collection-specific, so users are created for the entire Solr cluster. If users need to be restricted to a specific collection, that can be done with the authorization rules.

Add a User or Edit a Password

The `set-user` command allows you to add users and change their passwords. For example, the following defines two users and their passwords:

```
curl --user solr:SolrRocks http://localhost:8983/solr/admin/authentication -H
'Content-type:application/json' -d '{
  "set-user": {"tom" : "TomIsCool" ,
               "harry":"HarrysSecret"}}'
```

Delete a User

The `delete-user` command allows you to remove a user. The user password does not need to be sent to remove a user. In the following example, we've asked that user IDs 'tom' and 'harry' be removed from the system.

```
curl --user solr:SolrRocks http://localhost:8983/solr/admin/authentication -H
'Content-type:application/json' -d  '{
 "delete-user": ["tom","harry"]}'
```

Set a property

Set arbitrary properties for authentication plugin. The only supported property is `'blockUnknown'`

```
curl --user solr:SolrRocks http://localhost:8983/solr/admin/authentication -H
'Content-type:application/json' -d  '{
 "set-property": {"blockUnknown":false}}'
```

Using BasicAuth with SolrJ

In SolrJ the basic authentication credentials need to be set for each request as in this example:

```
SolrRequest req ;//create a new request object
req.setBasicAuthCredentials(userName, password);
solrClient.request(req);
```

Using command line scripts with BasicAuth

Add the following line to the `solr.in.sh` file. This example sets BasicAuth credentials for user-name "solr" and password "SolrRocks":

```
SOLR_AUTHENTICATION_OPTS="-Dbasicauth=solr:SolrRocks"
```

Securing inter-node requests

There are a lot of requests that originate from the Solr nodes itself. e.g: requests from overseer to nodes, recovery threads etc . These requests do not carry any basic auth credentials because no user initiated these requests. This means the user is Solr itself. Solr uses a special internode authentication mechanism where each Solr node is a super user and is fully trusted by other Solr nodes.

PKIAuthenticationPlugin

This kicks in when there is any request going on between 2 Solr nodes. It is enabled only when the Authentication plugin does not wish to handle inter-node security (only BasicAuthPlugin as of now) .For each outgoing request `PKIAuthenticationPlugin` adds a special header `'SolrAuth'` which carries the timestamp and principal encrypted using the private key of that node. The public key is exposed through an API so that any node can read it whenever it needs it. Any node who gets the request with that header, would get the public key from the sender and decrypt the information. if it is able to decrypt the data, the request trusted. It is invalid if the timestamp is more than 5 secs old. This assumes that the clocks of different nodes in the cluster are synchronized. The timeout is configurable through a system property called 'pkiauth.ttl'. For example , if you wish to bump up the ttl to 10 seconds (10000 milliseconds) , start each node with a property `'-Dpkiauth.ttl=10000'` .

Kerberos Authentication Plugin

If you are using Kerberos to secure your network environment, the Kerberos authentication plugin can be used to secure a Solr cluster. This allows Solr to use a Kerberos service principal and keytab file to authenticate with ZooKeeper and between nodes of the Solr cluster. Users of the Admin UI and alll clients (such as SolrJ) would also need to have a valid ticket before being able to use the UI or send requests to Solr.

Support for the Kerberos authentication plugin is only available in SolrCloud mode.



If you are using Solr with a Hadoop cluster secured with Kerberos and intend to store your Solr indexes in HDFS, also see the section Running Solr on HDFS for additional steps to configure Solr for that

purpose. The instructions on this page apply only to scenarios where Solr will be secured with Kerberos. If you only need to store your indexes in a Kerberized HDFS system, please see the other section referenced above.

How Solr Works With Kerberos

When setting up Solr to use Kerberos, configurations are put in place for Solr to use a *service principal*, or a Kerberos username, which is registered with the Key Distribution Center (KDC) to authenticate requests. The configurations define the service principal name and the location of the keytab file that contains the credentials.

security.json

The Solr authentication model uses a file called `/security.json` which is stored in ZooKeeper. A description of this file and how it is created and maintained is covered in the section Authentication and Authorization Plugins, and can only be used when Solr is running in SolrCloud mode. If this file is created after an initial startup of Solr, a restart of the system on each node is required.

Alternatively, the authentication plugin implementation can be specified during node startup using the system parameter: `-DauthenticationPlugin=org.apache.solr.security.KerberosPlugin`. This parameter can be used with either SolrCloud mode or standalone mode. However, if you are using Solr in standalone mode, this system parameter is the only way to enable Kerberos.

If you are using SolrCloud mode, the approach to use `security.json` is the best practice.

Service Principals and Keytab Files

Each Solr node must have a service principal registered with the Key Distribution Center (KDC). The Kerberos plugin uses SPNego to negotiate authentication.

Using `HTTP/host1@YOUR-DOMAIN.ORG`, as an example of a service principal:

- `HTTP` indicates the type of requests which this service principal will be used to authenticate. The `HTTP/` in the service principal is a must for SPNego to work with requests to Solr over HTTP.
- `host1` is the host name of the machine hosting the Solr node.
- `YOUR-DOMAIN.ORG` is the organization wide Kerberos realm.

Multiple Solr nodes on the same host may have the same service principal, since the host name is common to them all.

Along with the service principal, each Solr node needs a keytab file which should contain the credentials of the service principal used. A keytab file contains encrypted credentials to support passwordless logins while obtaining Kerberos tickets from the KDC. For each Solr node, the keytab file should be kept in a secure location and not shared with users of the cluster.

Since a Solr cluster requires internode communication, each node must also be able to make Kerberos enabled requests to other nodes. By default, Solr uses the same service principal and keytab as a 'client principal' for internode communication. You may configure a distinct client principal explicitly, but doing so is not recommended and is not covered in the examples below.

Kerberized ZooKeeper

When setting up a kerberized Solr cluster, it is recommended to enable Kerberos security for Zookeeper as well. In such a setup, the client principal used to authenticate requests with Zookeeper can be shared for internode communication as well. This has the benefit of not needing to renew the ticket granting tickets (TGTs) separately, since the Zookeeper client used by Solr takes care of this. To achieve this, a single JAAS configuration (with the app name as Client) can be used for the Kerberos plugin as well as for the Zookeeper client. See the configuration section below for an example of starting Zookeeper in Kerberos mode.

Browser Configuration

In order for your browser to access the Solr Admin UI after enabling Kerberos authentication, it must be able to negotiate with the Kerberos authenticator service to allow you access. Each browser supports this differently, and some (like Chrome) do not support it at all. If you see 401 errors when trying to access the Solr Admin UI after enabling Kerberos authentication, it's likely your browser has not been configured properly to know how or where to negotiate the authentication request.

Detailed information on how to set up your browser is beyond the scope of this documentation; please see your system administrators for Kerberos for details on how to configure your browser.

Plugin Configuration

Consult Your Kerberos Admins!
Before attempting to configure Solr to use Kerberos authentication, please review each step outlined below and consult with your local Kerberos administrators on each detail to be sure you know the correct values for each parameter. Small errors can cause Solr to not start or not function properly, and are notoriously difficult to diagnose.

Configuration of the Kerberos plugin has several parts:

- Create service principals and keytab files
- ZooKeeper configuration
- Create or update `/security.json`
- Define `jaas-client.conf`
- Solr startup parameters

We'll walk through each of these steps below.

Using Hostnames
To use host names instead of IP addresses, use the SOLR_HOST config in `bin/solr.in.sh` or pass a `-Dhost=<hostname>` during Solr startup. This guide uses IP addresses . If you specify a hostname replace all the IP addresses in the guide with the solr hostname

Get Service Principals and Keytabs

Before configuring Solr, make sure you have a Kerberos service principal for each Solr host and ZooKeeper (if ZooKeeper has not already been configured) available in the KDC server, and generate a keytab file as shown below.

This example assumes the hostname is `192.168.0.107` and your home directory is `/home/foo/`. This example should be modified for your own environment.

```
root@kdc:/# kadmin.local
Authenticating as principal foo/admin@EXAMPLE.COM with password.

kadmin.local:  addprinc HTTP/192.168.0.107
WARNING: no policy specified for HTTP/192.168.0.107@EXAMPLE.COM; defaulting to no
policy
Enter password for principal "HTTP/192.168.0.107@EXAMPLE.COM":
Re-enter password for principal "HTTP/192.168.0.107@EXAMPLE.COM":
Principal "HTTP/192.168.0.107@EXAMPLE.COM" created.

kadmin.local:  ktadd -k /tmp/107.keytab HTTP/192.168.0.107
Entry for principal HTTP/192.168.0.107 with kvno 2, encryption type
aes256-cts-hmac-sha1-96 added to keytab WRFILE:/tmp/107.keytab.
Entry for principal HTTP/192.168.0.107 with kvno 2, encryption type arcfour-hmac
added to keytab WRFILE:/tmp/107.keytab.
Entry for principal HTTP/192.168.0.107 with kvno 2, encryption type des3-cbc-sha1
added to keytab WRFILE:/tmp/108.keytab.
Entry for principal HTTP/192.168.0.107 with kvno 2, encryption type des-cbc-crc
added to keytab WRFILE:/tmp/107.keytab.

kadmin.local:  quit
```

Copy the keytab file from the KDC server's `/tmp/107.keytab` location to the Solr host at `/keytabs/107.keytab`. Repeat this step for each Solr node.

You might need to take similar steps to create a Zookeeper service principal and keytab if it has not already been set up. In that case, the example below shows a different service principal for ZooKeeper, so the above might be repeated with `zookeeper/host1` as the service principal for one of the nodes

ZooKeeper Configuration

If you are using a ZooKeeper that has already been configured to use Kerberos, you can skip the ZooKeeper-related steps shown here.

Since ZooKeeper manages the communication between nodes in a SolrCloud cluster, it must also be able to authenticate with each node of the cluster. Configuration requires setting up a service principal for ZooKeeper, defining a JAAS configuration file and instructing ZooKeeper to use both of those items.

The first step is to create a file `java.env` in ZooKeeper's `conf` directory and add the following to it, as in this example:

```
export
JVMFLAGS="-Djava.security.auth.login.config=/etc/zookeeper/conf/jaas-client.conf"
```

The JAAS configuration file should contain the following parameters. Be sure to change the `principal` and `keyTab` path as appropriate. The file must be located in the path defined in the step above, with the filename specified.

```
Server {
 com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="/keytabs/zkhost1.keytab"
  storeKey=true
  doNotPrompt=true
  useTicketCache=false
  debug=true
  principal=”zookeeper/host1”;
};
```

Finally, add the following lines to the ZooKeeper configuration file `zoo.cfg`:

```
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
jaasLoginRenew=3600000
```

Once all of the pieces are in place, start ZooKeeper with the following parameter pointing to the JAAS configuration file:

```
bin/zkServer.sh start
-Djava.security.auth.login.config=/etc/zookeeper/conf/jaas-client.conf
```

Create /security.json

Set up Solr to use the Kerberos plugin by uploading the `security.json` as follows:

```
> server/scripts/cloud-scripts/zkcli.sh -zkhost localhost:2181 -cmd put
/security.json '{"authentication":{"class":
"org.apache.solr.security.KerberosPlugin"}}'
```

More details on how to use a `/security.json` file in Solr are available in the section Authentication and Authorization Plugins.



If you already have a `/security.json` file in Zookeeper, download the file, add or modify the authentication section and upload it back to ZooKeeper using the Command Line Utilities available in Solr.

Define a JAAS Configuration File

The JAAS configuration file defines the properties to use for authentication, such as the service principal and the location of the keytab file. Other properties can also be set to ensure ticket caching and other features.

The following example can be copied and modified slightly for your environment. The location of the file can be anywhere on the server, but it will be referenced when starting Solr so it must be readable on the filesystem. The JAAS file may contain multiple sections for different users, but each section must have a unique name so it can be uniquely referenced in each application.

In the below example, we have created a JAAS configuration file with the name and path of `/home/foo/jaas-client.conf`. We will use this name and path when we define the Solr start parameters in the next section. Note that the client `principal` here is the same as the service principal. This will be used to authenticate internode requests and requests to Zookeeper. Make sure to use the correct `principal` hostname and the `keyTab` file path.

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="/keytabs/107.keytab"
  storeKey=true
  useTicketCache=true
  debug=true
  principal="HTTP/192.168.0.107@EXAMPLE.COM";
};
```

The first line of this file defines the section name, which will be used with the `solr.kerberos.jaas.appname` parameter, defined below.

The main properties we are concerned with are the `keyTab` and `principal` properties, but there are others which may be required for your environment. The javadocs for the Krb5LoginModule (the class that's being used and is called in the second line above) provide a good outline of the available properties, but for reference the ones in use in the above example are explained here:

- `useKeyTab`: this boolean property defines if we should use a keytab file (true, in this case).
- `keyTab`: the location and name of the keytab file for the principal this section of the JAAS configuration file is for. The path should be enclosed in double-quotes.
- `storeKey`: this boolean property allows the key to be stored in the private credentials of the user.
- `useTicketCache`: this boolean property allows the ticket to be obtained from the ticket cache.
- `debug`: this boolean property will output debug messages for help in troubleshooting.
- `principal`: the name of the service principal to be used.

Solr Startup Parameters

While starting up Solr, the following host-specific parameters need to be passed. These parameters can be passed at the command line with the `bin/solr` start script (see Solr Start Script Reference for details on how to pass system parameters) or defined in `bin/solr.in.sh` or `bin/solr.in.cmd` as appropriate for your operating system.

Parameter Name	Required	Description
solr.kerberos.name.rules	No	Used to map Kerberos principals to short names. Default value is `DEFAULT`. Example of a name rule: `RULE:[1:$1@$0](.*EXAMPLE.COM)s/@.*//`
solr.kerberos.cookie.domain	Yes	Used to issue cookies and should have the hostname of the Solr node.
solr.kerberos.cookie.portaware	No	When set to true, cookies are differentiated based on host and port, as opposed to standard cookies which are not port aware. This should be set if more than one Solr node is hosted on the same host. The default is false.
solr.kerberos.principal	Yes	The service principal.
solr.kerberos.keytab	Yes	Keytab file path containing service principal credentials.
solr.kerberos.jaas.appname	No	The app name (section name) within the JAAS configuration file which is required for internode communication. Default is `Client`, which is used for Zookeeper authentication as well. If different users are used for ZooKeeper and Solr, they will need to have separate sections in the JAAS configuration file.

java.security.auth.login.config	Yes	Path to the JAAS configuration file for configuring a Solr client for internode communication.

Here is an example that could be added to `bin/solr.in.sh`. Make sure to change this example to use the right hostname and the keytab file path.

```
SOLR_AUTHENTICATION_CLIENT_CONFIGURER=org.apache.solr.client.solrj.impl.Krb5HttpClie
ntConfigurer
SOLR_AUTHENTICATION_OPTS="-Djava.security.auth.login.config=/home/foo/jaas-client.co
nf -Dsolr.kerberos.cookie.domain=192.168.0.107 -Dsolr.kerberos.cookie.portaware=true
-Dsolr.kerberos.principal=HTTP/192.168.0.107@EXAMPLE.COM
-Dsolr.kerberos.keytab=/keytabs/107.keytab"
```

KDC with AES-256 encryption
If your KDC uses AES-256 encryption, you need to add the Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files to your JRE before a kerberized Solr can interact with the KDC.

You will know this when you see an error like this in your Solr logs : "KrbException: Encryption type AES256 CTS mode with HMAC SHA1-96 is not supported/enabled"

For Java 1.8, this is available here: http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html.

Replace the `local_policy.jar` present in `JAVA_HOME/jre/lib/security/` with the new `local_policy.jar` from the downloaded package and restart the Solr node.

Using Delegation Tokens

The Kerberos plugin can be configured to use delegation tokens, which allow an application to reuse the authentication of an end-user or another application.

There are a few use cases for Solr where this might be helpful:

- Using distributed clients (such as MapReduce) where each client may not have access to the user's credentials.
- When load on the Kerberos server is high. Delegation tokens can reduce the load because they do not access the server after the first request.
- If requests or permissions need to be delegated to another user.

To enable delegation tokens, several parameters must be defined. These parameters can be passed at the command line with the `bin/solr` start script (see Solr Start Script Reference for details on how to pass system parameters) or defined in `bin/solr.in.sh` or `bin/solr.in.cmd` as appropriate for your operating system.

Parameter Name	Required	Description
solr.kerberos.delegation.token.enabled	Yes, to enable tokens	False by default, set to true to enable delegation tokens.
solr.kerberos.delegation.token.kind	No	Type of delegation tokens. By default this is `solr-dt`. Likely this does not need to change. No other option is available at this time.

solr.kerberos.delegation.token.validity	No	Time, in seconds, for which delegation tokens are valid. The default is 36000 seconds.
solr.kerberos.delegation.token.signer.secret.provider	No	Where delegation token information is stored internally. The default is `zookeeper` which must be the location for delegation tokens to work across Solr servers (when running in SolrCloud mode). No other option is available at this time.
solr.kerberos.delegation.token.signer.secret.provider.zookeper.path	No	The ZooKeeper path where the secret provider information is stored. This is in the form of the chrooted path + /security/token. For example, `server1:9983,server2:9983,server3:9983/solr/security/token`.
solr.kerberos.delegation.token.secret.manager.znode.working.path	No	The ZooKeeper path where token information is stored. This is in the form of the chrooted path + /security/zkdtsm. For example, `server1:9983,server2:9983,server3:9983/solr/security/zkdtsm`.

Start Solr

Once the configuration is complete, you can start Solr with the `bin/solr` script, as in the example below. This example assumes you modified `bin/solr.in.sh` or `bin/solr.in.cmd`, with the proper values, but if you did not, you would pas the system parameters along with the start command. Note you also need to customize the `-z` property as appropriate for the location of your ZooKeeper nodes.

```
bin/solr -c -z server1:2181,server2:2181,server3:2181/solr
```

Test the Configuration

1. Do a `kinit` with your username. For example, "`kinit user@EXAMPLE.COM`".
2. Try to access Solr using `curl`. You should get a successful response.

```
curl --negotiate -u : "http://192.168.0.107:8983/solr/"
```

Using SolrJ with a Kerberized Solr

To use Kerberos authentication in a SolrJ application, you need the following two lines before you create a SolrClient:

```
System.setProperty("java.security.auth.login.config", "/home/foo/jaas-client.conf");
HttpClientUtil.setConfigurer(new Krb5HttpClientConfigurer());
```

You need to specify a Kerberos service principal for the client and a corresponding keytab in the JAAS client configuration file above. This principal should be different from the service principal we created for Solr .

Here's an example:

```
SolrJClient {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  keyTab="/keytabs/foo.keytab"
  storeKey=true
  useTicketCache=true
  debug=true
  principal="solrclient@EXAMPLE.COM";
};
```

Delegation tokens are also supported with SolrJ, in the following ways:

- `DelegationTokenRequest` and `DelegationTokenResponse` can be used to get, cancel, and renew delegation tokens.
- `HttpSolrClient.Builder` includes a `withDelegationToken` function for creating an HttpSolrClient that uses a delegation token to authenticate.

Hadoop's delegation token responses are in JSON map format. A response parser for that is available in `DelegationTokenResponse`. Other response parsers may not work well with Hadoop responses.

Rule-Based Authorization Plugin

Solr allows configuring roles to control user access to the system. This is accomplished through rule-based permission definitions which are assigned to users. The roles are fully customizable, and provide the ability to limit access to specific collections, request handlers, request parameters, and request methods.

The roles can be used with any of the authentication plugins or with a custom authentication plugin if you have created one. You will only need to ensure that you configure the role-to-user mappings with the proper user IDs that your authentication system provides.

Once defined through the API, roles are stored in `security.json` in ZooKeeper. This means this feature is available **when using Solr in SolrCloud mode only**.

Enable the Authorization Plugin

The plugin must be enabled in `security.json`. This file and how to upload it to ZooKeeper is described in detail in the section Enable Plugins with security.json.

This file has two parts, the `authentication` part and the `authorization` part. The `authentication` part stores information about the class being used for authentication.

The `authorization` part is not related to Basic authentication, but is a separate authorization plugin designed to support fine-grained user access control. When creating `security.json` you can add the permissions to the file, or you can use the Authorization API described below to add them as needed.

This example `security.json` shows how the Basic authentication plugin can work with this authorization plugin:

```
{
"authentication":{
   "class":"solr.BasicAuthPlugin",
   "blockUnknown": true,
   "credentials":{"solr":"IV0EHq1OnNrj6gvRCwvFwTrZ1+z1oBbnQdiVC3otuq0=
Ndd7LKvVBAaZIF0QAVi1ekCfAJXr1GGfLtRUXhgrF8c="}
},
"authorization":{
   "class":"solr.RuleBasedAuthorizationPlugin",
   "permissions":[{"name":"security-edit",
      "role":"admin"}]
   "user-role":{"solr":"admin"}
}}
```

There are several things defined in this example:

- Basic authentication and rule-based authorization plugins are enabled.
- A user called 'solr', with a password has been defined.
- All requests w/o credentials will be rejected with a 401 error. Set `'blockUnknown'` to false (or remove it altogether) if you wish to let unauthenticated requests to go through. However, if a particular resource is protected by a rule, they are rejected anyway with a 401 error.
- The 'admin' role has been defined, and it has permission to edit security settings.
- The 'solr' user has been defined to the 'admin' role.

Permission Attributes

Each role is comprised of one or more permissions which define what the user is allowed to do. Each permission is made up of several attributes that define the allowed activity. There are some pre-defined permissions which cannot be modified.

The permissions are consulted in order they appear in `security.json`. The first permission that matches is applied for each user, so the strictest permissions should be at the top of the list. Permissions order can be controlled with a parameter of the Authorization API, as described below.

Predefined Permissions

There are several permissions that are pre-defined. These have fixed default values, which cannot be modified, and new attributes cannot be added. To use these attributes, simply define a role that includes this permission, and then assign a user to that role.

The pre-defined permissions are:

- **security-edit:** this permission is allowed to edit the security configuration, meaning any update action that modifies `security.json` through the APIs will be allowed.
- **security-read**: this permission is allowed to read the security configuration, meaning any action that reads `security.json` settings through the APIs will be allowed.
- **schema-edit**: this permission is allowed to edit a collection's schema using the Schema API. Note that this allows schema edit permissions for *all* collections. If edit permissions should only be applied to specific collections, a custom permission would need to be created.
- **schema-read**: this permission is allowed to read a collection's schema using the Schema API. Note that this allows schema read permissions for *all* collections. If read permissions should only be applied to specific collections, a custom permission would need to be created.
- **config-edit**: this permission is allowed to edit a collection's configuration using the Config API, the Request Parameters API, and other APIs which modify `configoverlay.json`. Note that this allows configuration edit permissions for *all* collections. If edit permissions should only be applied to specific collections, a custom permission would need to be created.
- **core-admin-read** : Read operations on the core admin API
- **core-admin-edit**: Core admin commands that can mutate the system state.

- **config-read**: this permission is allowed to read a collection's configuration using the Config API, the Request Parameters API, and other APIs which modify `configoverlay.json`. Note that this allows configuration read permissions for *all* collections. If read permissions should only be applied to specific collections, a custom permission would need to be created.
- **collection-admin-edit**: this permission is allowed to edit a collection's configuration using the Collections API. Note that this allows configuration edit permissions for *all* collections. If edit permissions should only be applied to specific collections, a custom permission would need to be created. Specifically, the following actions of the Collections API would be allowed:
    - CREATE
    - RELOAD
    - SPLITSHARD
    - CREATESHARD
    - DELETESHARD
    - CREATEALIAS
    - DELETEALIAS
    - DELETE
    - DELETEREPLICA
    - ADDREPLICA
    - CLUSTERPROP
    - MIGRATE
    - ADDROLE
    - REMOVEROLE
    - ADDREPLICAPROP
    - DELETEREPLICAPROP
    - BALANCESHARDUNIQUE
    - REBALANCELEADERS
- **collection-admin-read**: this permission is allowed to read a collection's configuration using the Collections API. Note that this allows configuration read permissions for *all* collections. If read permissions should only be applied to specific collections, a custom permission would need to be created. Specifically, the following actions of the Collections API would be allowed:
    - LIST
    - OVERSEERSTATUS
    - CLUSTERSTATUS
    - REQUESTSTATUS
- **update**: this permission is allowed to perform any update action on any collection. This includes sending documents for indexing (using an update request handler).
- **read**: this permission is allowed to perform any read action on any collection. This includes querying using search handlers (using request handlers) such as `/select`, `/get`, `/browse`, `/tvrh`, `/terms`, `/clustering`, `/elevate`, `/export`, `/spell`, `/clustering`, and `/sql`.
- **all**: Any requests coming to Solr.

Authorization API

API Endpoint

`/admin/authorization`: takes a set of commands to create permissions, map permissions to roles, and map roles to users.

Manage Permissions

Three commands control managing permissions:

- `set-permission`: create a new permission, overwrite an existing permission definition, or assign a pre-defined permission to a role.
- `update-permission`: update some attributes of an existing permission definition.
- `delete-permission`: remove a permission definition.

Permissions need to be created if they are not on the list of pre-defined permissions above.

Several properties can be used to define your custom permission.

Property	Description
name	The name of the permission. This is required only if it is a predefined permission.
collection	The collection or collections the permission will apply to. When the path that will be allowed is collection-specific, such as when setting permissions to allow useof the Schema API, omitting the collection property will allow the defined path and/or method for all collections. However, when the path is one that is non-collection-specific, such as the Collections API, the collection value must be `null`.
path	A request handler name, such as `/update` or `/select`. A wild card is supported, to allow for all paths as appropriate (such as, `/update/*`).
method	HTTP methods that are allowed for this permission. You could allow only GET requests, or have a role that allows PUT and POST requests. The method values that are allowed for this property are GET, POST, PUT,DELETEand HEAD.

params	The names and values of request parameters. This property can be omitted if all request parameters are to be matched, but will restrict access only to the values provided if defined.
	
	For example, this property could be used to limit the actions a role is allowed to perform with the Collections API. If the role should only be allowed to perform the LIST or CLUSTERSTATUS requests, you would define this as follows:
	
	`"params": {`
	`"action": [LIST, CLUSTERSTATUS]`
	`}`
	
	The value of the parameter can be a simple string or it could be a regular expression. use the prefix `REGEX:` to use a regular expression match instead of a string identity match
	
	If the commands LIST and CLUSTERSTATUS are case insensitive, the above example should be as follows
	
	`"params": {`
	`"action": ["REGEX:(?i)LIST", "REGEX:(?i)CLUSTERSTATUS"]`
	`}`

before	This property allows ordering of permissions. The value of this property is the index of the permission that this new permission should be placed before in `security.json`. The index is automatically assigned in the order they are created
role	The name of the role(s) to give this permission. This name will be used to map user IDs to the role to grant these permissions. The value can be wildcard such as (*), which means that any user is OK, but no user is NOT OK.

The following would create a new permission named "collection-mgr" that is allowed to create and list collections. The permission will be placed before the "read" permission. Note also that we have defined "collection as `null`, this is because requests to the Collections API are never collection-specific.

```
curl --user solr:SolrRocks -H 'Content-type:application/json' -d '{
  "set-permission": {"collection": null,
                     "path":"/admin/collections",
                     "params":{"action":[LIST, CREATE]},
                     "before: 3,
                     "role": "admin"}
}' http://localhost:8983/solr/admin/authorization
```

update or delete permissions

Permissions can be accessed using their index in the list. Use the GET /security/authorization to see the existing permissions and their indices.

the following example updates the `'role'` attribute of permission at index `'3'`

```
curl --user solr:SolrRocks -H 'Content-type:application/json' -d '{
  "update-permission": {"index": 3,
                        "role": ["admin", "dev"]}
}' http://localhost:8983/solr/admin/authorization
```

the following example deletes permission at index `'3'`

```
curl --user solr:SolrRocks -H 'Content-type:application/json' -d '{
  "delete-permission": 3
}' http://localhost:8983/solr/admin/authorization
```

Map Roles to Users

A single command allows roles to be mapped to users:

- `set-user-role`: map a user to a permission.

To remove a user's permission, you should set the role to `null`. There is no command to delete a user role.

The values supplied to the command are simply a user ID and one or more roles the user should have.

For example, the following would grant a user "solr" the "admin" and "dev" roles, and remove all roles from the user ID "harry":

```
curl -u solr:SolrRocks -H 'Content-type:application/json' -d '{
   "set-user-role" : {"solr": ["admin","dev"],
                      "harry": null}
}' http://localhost:8983/solr/admin/authorization
```

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
  - Convert the certificate and key to PEM format for use with cURL
  - Set common SSL related system properties
  - Run Single Node Solr using SSL
- SolrCloud
  - Configure ZooKeeper
  - Run SolrCloud with SSL
- Example Client Actions
  - Create a SolrCloud collection using bin/solr
  - Retrieve SolrCloud cluster status using cURL
  - Index documents using post.jar
  - Query using cURL
  - Index a document using CloudSolrClient

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 `server/etc/` directory in the binary Solr distribution. It's assumed that you have the JDK `keytool` utility on your `PATH`, and that `openssl` is also on your `PATH`. See https://www.openssl.org/related/binaries.html for OpenSSL binaries for Windows and Solaris.

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
```

The keytool application will prompt you to create a destination keystore password and for the source keystore password, which was set when creating the keystore ("secret" in the example shown above).

Next convert the PKCS12 format keystore, including both the certificate and the key, into PEM format using the `o`

`penssl` command:

```
openssl pkcs12 -in solr-ssl.keystore.p12 -out solr-ssl.pem
```

If you want to use cURL on OS X Yosemite (10.10), you'll need to create a certificate-only version of the PEM format, as follows:

```
openssl pkcs12 -nokeys -in solr-ssl.keystore.p12 -out solr-ssl.cacert.pem
```

Set common SSL related system properties

The Solr start script is already setup to pass SSL-related Java system properties to the JVM. To activate the SSL settings, uncomment and update the set of properties beginning with SOLR_SSL_* in `bin/solr.in.sh`. (or `bin\solr.in.cmd` on Windows). Note, if you setup Solr as a service on Linux using the steps outlined in Taking Solr to Production, then make these changes in `/var/solr/solr.in.sh` instead.

bin/solr.in.sh example SOLR_SSL_* configuration

```
SOLR_SSL_KEY_STORE=etc/solr-ssl.keystore.jks
SOLR_SSL_KEY_STORE_PASSWORD=secret
SOLR_SSL_TRUST_STORE=etc/solr-ssl.keystore.jks
SOLR_SSL_TRUST_STORE_PASSWORD=secret
# Require clients to authenticate
SOLR_SSL_NEED_CLIENT_AUTH=false
# Enable clients to authenticate (but not require)
SOLR_SSL_WANT_CLIENT_AUTH=false
```

When you start Solr, the `bin/solr` script includes the settings in `bin/solr.in.sh` and will pass these SSL-related system properties to the JVM.

Client Authentication Settings
Enable either SOLR_SSL_NEED_CLIENT_AUTH or SOLR_SSL_WANT_CLIENT_AUTH but not both at the same time. They are mutually exclusive and Jetty will select one of them which may not be what you expect.

Similarly, when you start Solr on Windows, the `bin\solr.cmd` script includes the settings in `bin\solr.in.cmd` - uncomment and update the set of properties beginning with `SOLR_SSL_*` to pass these SSL-related system properties to the JVM:

bin\solr.in.cmd example SOLR_SSL_* configuration

```
set SOLR_SSL_KEY_STORE=etc/solr-ssl.keystore.jks
set SOLR_SSL_KEY_STORE_PASSWORD=secret
set SOLR_SSL_TRUST_STORE=etc/solr-ssl.keystore.jks
set SOLR_SSL_TRUST_STORE_PASSWORD=secret
REM Require clients to authenticate
set SOLR_SSL_NEED_CLIENT_AUTH=false
REM Enable clients to authenticate (but not require)
set SOLR_SSL_WANT_CLIENT_AUTH=false
```

Run Single Node Solr using SSL

Start Solr using the command shown below; by default clients will not be required to authenticate:

***nix command**

```
bin/solr -p 8984
```

Windows command

```
bin\solr.cmd -p 8984
```

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` tool that comes with the binary Solr distribution to do this:

***nix command**

```
server/scripts/cloud-scripts/zkcli.sh -zkhost localhost:2181 -cmd clusterprop -name
urlScheme -val https
```

Windows command

```
server\scripts\cloud-scripts\zkcli.bat -zkhost localhost:2181 -cmd clusterprop -name
urlScheme -val https
```

If you have set up your ZooKeeper cluster to use a chroot for Solr, make sure you use the correct `zkhost` string with `zkcli`, e.g. `-zkhost localhost:2181/solr`.

Run SolrCloud with SSL

Create Solr home directories for two nodes

Create two copies of the `server/solr/` directory which will serve as the Solr home directories for each of your two SolrCloud nodes:

***nix commands**

```
mkdir cloud
cp -r server/solr cloud/node1
cp -r server/solr cloud/node2
```

Windows commands

```
mkdir cloud
xcopy /E server\solr cloud\node1\
xcopy /E server\solr cloud\node2\
```

Start the first Solr node

Next, start the first Solr node on port 8984. Be sure to stop the standalone server first if you started it when working through the previous section on this page.

***nix command**

```
bin/solr -cloud -s cloud/node1 -z localhost:2181 -p 8984
```

Windows command

```
bin\solr.cmd -cloud -s cloud\node1 -z localhost:2181 -p 8984
```

Notice the use of the `-s` option to set the location of the Solr home directory for node1.

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`:

***nix command**

```
bin/solr -cloud -s cloud/node1 -z localhost:2181 -p 8984
-Dsolr.ssl.checkPeerName=false
```

Windows command

```
bin\solr.cmd -cloud -s cloud\node1 -z localhost:2181 -p 8984
-Dsolr.ssl.checkPeerName=false
```

Start the second Solr node

Finally, start the second Solr node on port 7574 - again, to skip hostname verification, add `-Dsolr.ssl.checkPeerName=false`;

***nix command**

```
bin/solr -cloud -s cloud/node2 -z localhost:2181 -p 7574
```

Windows command

```
bin\solr.cmd -cloud -s cloud\node2 -z localhost:2181 -p 7574
```

Example Client Actions

cURL on OS X Mavericks (10.9) 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 . cURL on OS X Yosemite (10.10) is improved - 2-way SSL is possible - see http://curl.haxx.se/mail/archive-2014-10/0053.html .

The cURL commands in the following sections will not work with the system `curl` on OS X Yosemite (10.10). Instead, the certificate supplied with the `-E` param must be in PKCS12 format, and the file supplied with the `--cacert` param must contain only the CA certificate, and no key (see above for instructions on creating this file):

```
curl -E solr-ssl.keystore.p12:secret --cacert solr-ssl.cacert.pem ...
```

If your operating system does not include cURL, you can download binaries here: http://curl.haxx.se/download.html

Create a SolrCloud collection using `bin/solr`

Create a 2-shard, replicationFactor=1 collection named mycollection using the default configset (data_driven_schema_configs):

***nix command**

```
bin/solr create -c mycollection -shards 2
```

Windows command

```
bin\solr.cmd create -c mycollection -shards 2
```

The `create` action will pass the `SOLR_SSL_*` properties set in your include file to the SolrJ code used to create the collection.

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=../../server/etc/solr-ssl.keystore.jks
-Djavax.net.ssl.trustStore=../../server/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 `CloudSolrClient`

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";
CloudSolrClient client = new CloudSolrClient.Builder().withZkHost(zkHost).build();
client.setDefaultCollection("mycollection");
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "1234");
doc.addField("name", "A lovely summer holiday");
client.add(doc);
client.commit();
```

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 MapReduce to process Solr data, rather it only uses the HDFS filesystem for index and transaction log file storage. To use Hadoop MapReduce to process Solr data, see the MapReduceIndexerTool in the Solr contrib area.

To use HDFS rather than a local filesystem, you must be using Hadoop 2.x and you will need to instruct Solr to use the `HdfsDirectoryFactory`. There are also several additional parameters to define. These can be set in one of three ways:

- Pass JVM arguments to the `bin/solr` script. These would need to be passed every time you start Solr with `bin/solr`.
- Modify `solr.in.sh` (or `solr.in.cmd` on Windows) to pass the JVM arguments automatically when using `bin/solr` without having to set them manually.
- Define the properties in `solrconfig.xml`. These configuration changes would need to be repeated for every collection, so is a good option if you only want some of your collections stored in HDFS.

Starting Solr on HDFS

Standalone Solr Instances

For standalone Solr instances, there are a few parameters you should be sure to modify before starting Solr. These can be set in `solrconfig.xml`(more on that below), or passed to the `bin/solr` script at startup.

- 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.

If you do not modify `solrconfig.xml`, you can instead start Solr on HDFS with the following command:

```
bin/solr start -Dsolr.directoryFactory=HdfsDirectoryFactory
     -Dsolr.lock.type=hdfs
     -Dsolr.data.dir=hdfs://host:port/path
     -Dsolr.updatelog=hdfs://host:port/path
```

This example will start Solr in standalone mode, using the defined JVM properties (explained in more detail below).

SolrCloud Instances

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.

```
bin/solr start -c -Dsolr.directoryFactory=HdfsDirectoryFactory
     -Dsolr.lock.type=hdfs
     -Dsolr.hdfs.home=hdfs://host:port/path
```

This command starts Solr in SolrCloud mode, using the defined JVM properties.

Modifying solr.in.sh (*nix) or solr.in.cmd (Windows)

The examples above assume you will pass JVM arguments as part of the start command every time you use `bin/solr` to start Solr. However, `bin/solr` looks for an include file named `solr.in.sh` (`solr.in.cmd` on Windows) to set environment variables. By default, this file is found in the `bin` directory, and you can modify it to permanently add the `HdfsDirectoryFactory` settings and ensure they are used every time Solr is started.

For example, to set JVM arguments to always use HDFS when running in SolrCloud mode (as shown above), you would add a section such as this:

```
# Set HDFS DirectoryFactory & Settings
-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
```

HdfsDirectoryFactory Parameters

The `HdfsDirectoryFactory` has a number of settings that are defined as part of the `directoryFactory` configuration.

Solr HDFS Settings

Parameter	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

Parameter	Default	Description
`solr.hdfs.blockcache.enabled`	true	Enable the blockcache
`solr.hdfs.blockcache.read.enabled`	true	Enable the read 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`	true	Enable/Disable using one global cache for all SolrCores. The settings used will be from the first HdfsDirectoryFactory created.

NRTCachingDirectory Settings

Parameter	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.

Parameter	Default	Description
`solr.hdfs.confdir`	N/A	Pass the location of HDFS client configuration files - needed for HDFS HA for example.

Kerberos Authentication Settings

Hadoop can be configured to use the Kerberos protocol to verify user identity when trying to access core services like HDFS. If your HDFS directories are protected using Kerberos, then you need to configure Solr's HdfsDirectoryFactory to authenticate using Kerberos in order to read and write to HDFS. To enable Kerberos

authentication from Solr, you need to set the following parameters:

Parameter	Default	Description
`solr.hdfs.security.kerberos.enabled`	false	Set to true to enable Kerberos authentication
`solr.hdfs.security.kerberos.keytabfile`	N/A	A keytab file contains pairs of Kerberos principals and encrypted keys which allows for password-less authentication when Solr attempts to authenticate with secure Hadoop. This file will need to be present on all Solr servers at the same path provided in this parameter.
`solr.hdfs.security.kerberos.principal`	N/A	The Kerberos principal that Solr should use to authenticate to secure Hadoop; the format of a typical Kerberos V5 principal is: `primary/instance@realm`

Example

Here is a sample `solrconfig.xml` configuration for storing Solr indexes on HDFS:

```
<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.nrtcachingdirectory.enable">true</bool>
  <int name="solr.hdfs.nrtcachingdirectory.maxmergesizemb">16</int>
  <int name="solr.hdfs.nrtcachingdirectory.maxcachedmb">192</int>
</directoryFactory>
```

If using Kerberos, you will need to add the three Kerberos related properties to the `<directoryFactory>` element in solrconfig.xml, such as:

```
<directoryFactory name="DirectoryFactory" class="solr.HdfsDirectoryFactory">
   ...
  <bool name="solr.hdfs.security.kerberos.enabled">true</bool>
  <str name="solr.hdfs.security.kerberos.keytabfile">/etc/krb5.keytab</str>
  <str name="solr.hdfs.security.kerberos.principal">solr/admin@KERBEROS.COM</str>
</directoryFactory>
```

Automatically Add Replicas in SolrCloud

One benefit to running Solr in HDFS is the ability to automatically add new replicas when the Overseer notices that a shard has gone down. Because the "gone" index shards are stored in HDFS, the a new core will be created and the new core will point to the existing indexes in HDFS.

Collections created using `autoAddReplicas=true` on a shared file system have automatic addition of replicas

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.

Temporarily disable autoAddReplicas for the entire cluster

When doing offline maintenance on the cluster and for various other use cases where an admin would like to temporarily disable auto addition of replicas, the following APIs will disable and re-enable autoAddReplicas for **all collections in the cluster**:

Disable auto addition of replicas cluster wide by setting the cluster property `autoAddReplicas` to `false`:

```
http://localhost:8983/solr/admin/collections?action=CLUSTERPROP&name=autoAddReplicas
&val=false
```

Re-enable auto addition of replicas (for those collections created with autoAddReplica=true) by unsetting the `autoAddReplicas` cluster property (when no `val` param is provided, the cluster property is unset):

```
http://localhost:8983/solr/admin/collections?action=CLUSTERPROP&name=autoAddReplicas
```

Making and Restoring Backups

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.

Solr provides two approaches to backing up and restoring Solr cores or collections, depending on how you are running Solr. If you run SolrCloud, you will use the Collections API; if you run Solr in standalone mode, you will use the replication handler.

SolrCloud

Support for backups when running SolrCloud is provided with the Collections API. This allows the backups to be generated across multiple shards, and restored to the same number of shards and replicas as the original collection.

Two commands are available:

- `action=BACKUP`: This command backs up Solr indexes and configurations. More information is available in the section Backup Collection.
- `action=RESTORE`: This command restores Solr indexes and configurations. More information is available in the section Restore Collection.

Standalone Mode

Backups and restoration uses Solr's replication handler. Out of the box, Solr includes implicit support for replication so this API can be used. Configuration of the replication handler can, however, be customized by defining your own replication handler in `solrconfig.xml` . For details on configuring the replication handler, see the section Configuring the ReplicationHandler.

Backup API

The backup API requires sending a command to the `/replication` handler to back up the system.

You can trigger a back-up with an HTTP command like this (replace "gettingstarted" with the name of the core you are working with):

Backup API

```
curl http://localhost:8983/solr/gettingstarted/replication?command=backup
```

The backup command is an asynchronous call, and it will represent data from the latest index commit point. All indexing and search operations will continue to be executed against the index as usual.

Only one backup call can be made against a core at any point in time. While an ongoing backup operation is happening subsequent calls for restoring will throw an exception.

The backup request can also take the following additional parameters:

Parameter	Description
location	The path where the backup will be created. If the path is not absolute then the backup path will be relative to Solr's instance directory.
name	The snapshot will be created in a directory called `snapshot.<name>`. If a name is not specified then the directory name would have the following format: `snapshot.<yyyyMMdd HHmmssSSS>`
numberToKeep	The number of backups to keep. If `maxNumberOfBackups` has been specified on the replication handler in `solrconfig.xml`, `maxNumberOfBackups` is always used and attempts to use `numberToKeep` will cause an error. Also, this parameter is not taken into consideration if the backup name is specified. More information about `maxNumberOfBackups` can be found in the section Configuring the ReplicationHandler.
repository	The name of the repository to be used for the backup. If no repository is specified then the local filesystem repository will be used automatically.
commitName	The name of the commit which was used while taking a snapshot using the CREATESNAPSHOT command.

Backup Status

The backup operation can be monitored to see if it has completed by sending the `details` command to the `/replication` handler, as in this example:

Status API

```
http://localhost:8983/solr/gettingstarted/replication?command=details"
```

Output Snippet

```
<lst name="backup">
  <str name="startTime">Sun Apr 12 16:22:50 DAVT 2015</str>
  <int name="fileCount">10</int>
  <str name="status">success</str>
  <str name="snapshotCompletedAt">Sun Apr 12 16:22:50 DAVT 2015</str>
  <str name="snapshotName">my_backup</str>
</lst>
```

If it failed then a `snapShootException` will be sent in the response.

Restore API

Restoring the backup requires sending the `restore` command to the `/replication` handler, followed by the name of the backup to restore.

You can restore from a backup with a command like this:

Example Usage

```
http://localhost:8983/solr/gettingstarted/replication?command=restore&name=backup_name"
```

This will restore the named index snapshot into the current core. Searches will start reflecting the snapshot data once the restore is complete.

The restore request can also take these additional parameters:

Name	Description
location	The location of the backup snapshot file. If not specified, it looks for backups in Solr's data directory.
name	The name of the backed up index snapshot to be restored. If the name is not provided it looks for backups with `snapshot.<timestamp>` format in the location directory. It picks the latest timestamp backup in that case.
repository	The name of the repository to be used for the backup. If no repository is specified then the local filesystem repository will be used automatically.

The restore command is an asynchronous call. Once the restore is complete the data reflected will be of the backed up index which was restored.

Only one restore call can can be made against a core at one point in time. While an ongoing restore operation is happening subsequent calls for restoring will throw an exception.

Restore Status API

You can also check the status of a restore operation by sending the `restorestatus` command to the `/replication` handler, as in this example:

Status API

```
curl -XGET
http://localhost:8983/solr/gettingstarted/replication?command=restorestatus
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
  </lst>
  <lst name="restorestatus">
    <str name="snapshotName">snapshot.<name></str>
    <str name="status">success</str>
  </lst>
</response>
```

The status value can be "In Progress" , "success" or "failed". If it failed then an "exception" will also be sent in the response.

Create Snapshot API

The snapshot functionality is different from the backup functionality as the index files aren't copied anywhere. The index files are snapshotted in the same index directory and can be referenced while taking backups.

You can trigger a snapshot command with an HTTP command like this (replace "techproducts" with the name of the core you are working with):

Create Snapshot API

```
curl
http://localhost:8983/solr/admin/cores?action=CREATESNAPSHOT&core=techproducts&commi
tName=commit1
```

The list snapshot request parameters are:

Parameter	Description
commitName	Specify the commit name to store the snapshot as
core	name of the core to perform the snapshot on
async	Request ID to track this action which will be processed asynchronously

List Snapshot API

The list snapshot functionality lists all the taken snapshots for a particular core.

You can trigger a list snapshot command with an HTTP command like this (replace "techproducts" with the name of the core you are working with):

List Snapshot API

```
curl
http://localhost:8983/solr/admin/cores?action=LISTSNAPSHOTS&core=techproducts&commit
Name=commit1
```

The list snapshot request parameters are:

Parameter	Description
core	name of the core to whose snapshots we want to list
async	Request ID to track this action which will be processed asynchronously

Delete Snapshot API

The delete snapshot functionality deletes a particular snapshot for a particular core.

You can trigger a delete snapshot command with an HTTP command like this (replace "techproducts" with the name of the core you are working with):

Delete Snapshot API

```
curl
http://localhost:8983/solr/admin/cores?action=DELETESNAPSHOT&core=techproducts&commi
tName=commit1
```

The delete snapshot request parameters are:

Parameter	Description
commitName	Specify the commit name to be deleted
core	name of the core whose snapshot we want to delete
async	Request ID to track this action which will be processed asynchronously

Backup/Restore Storage Repositories

Solr provides interfaces to plug different storage systems for backing up and restoring. For example, you can have a Solr cluster running on a local filesystem like EXT3 but you can backup the indexes to a HDFS filesystem or vice versa.

The repository interfaces needs to be configured in the solr.xml file . While running backup/restore commands we can specify the repository to be used.

If no repository is configured then the local filesystem repository will be used automatically.

Example solr.xml section to configure a repository like HDFS .

Output

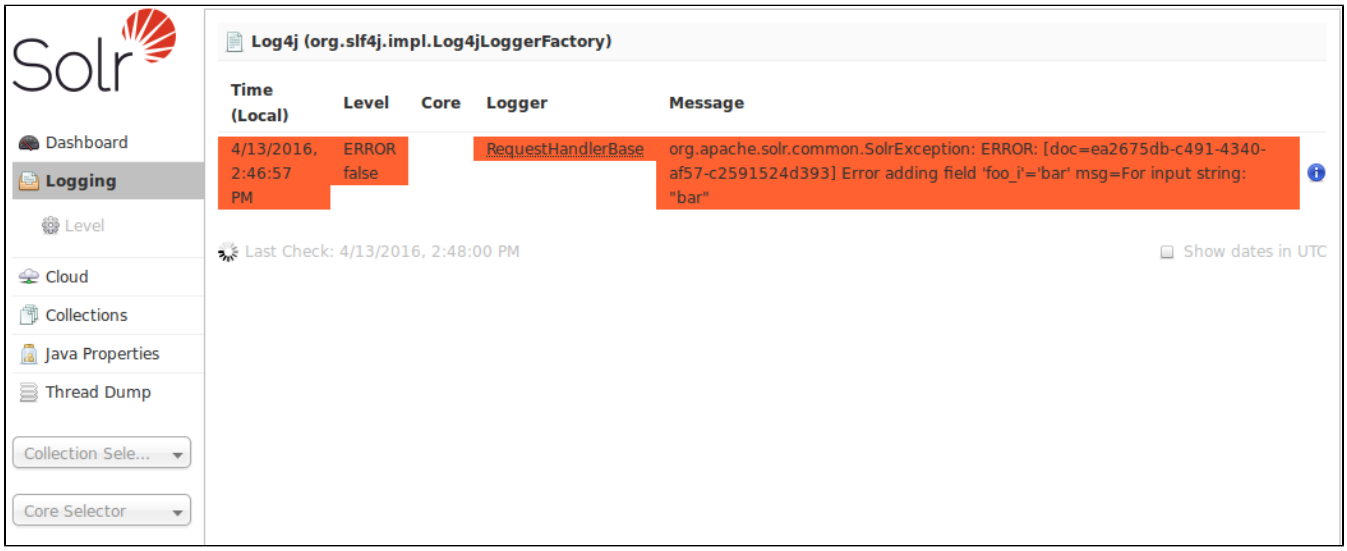
```
<backup>
 <repository name="hdfs"
class="org.apache.solr.core.backup.repository.HdfsBackupRepository"
default="false"/>
 <str name="location">${solr.hdfs.default.backup.path}</str>
 <str name="solr.hdfs.home">${solr.hdfs.home:}</str>
 <str name="solr.hdfs.confdir">${solr.hdfs.confdir:}</str>
</backup>
```

Configuring Logging

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

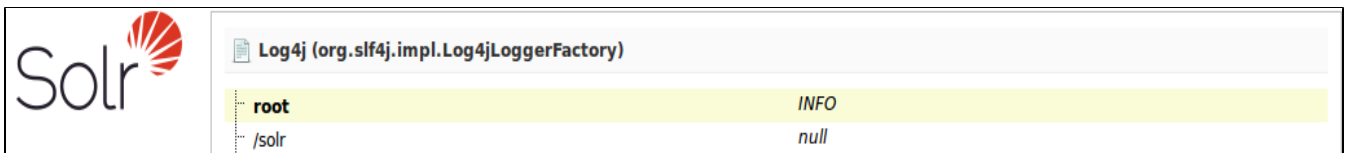
You can control the amount of logging output in Solr by using the Admin Web interface. Select the **LOGGING** lin k. Note that this page only lets you change settings in the running system and is not saved for the next run. (For more information about the Admin Web interface, see Using the Solr Administration User Interface.)

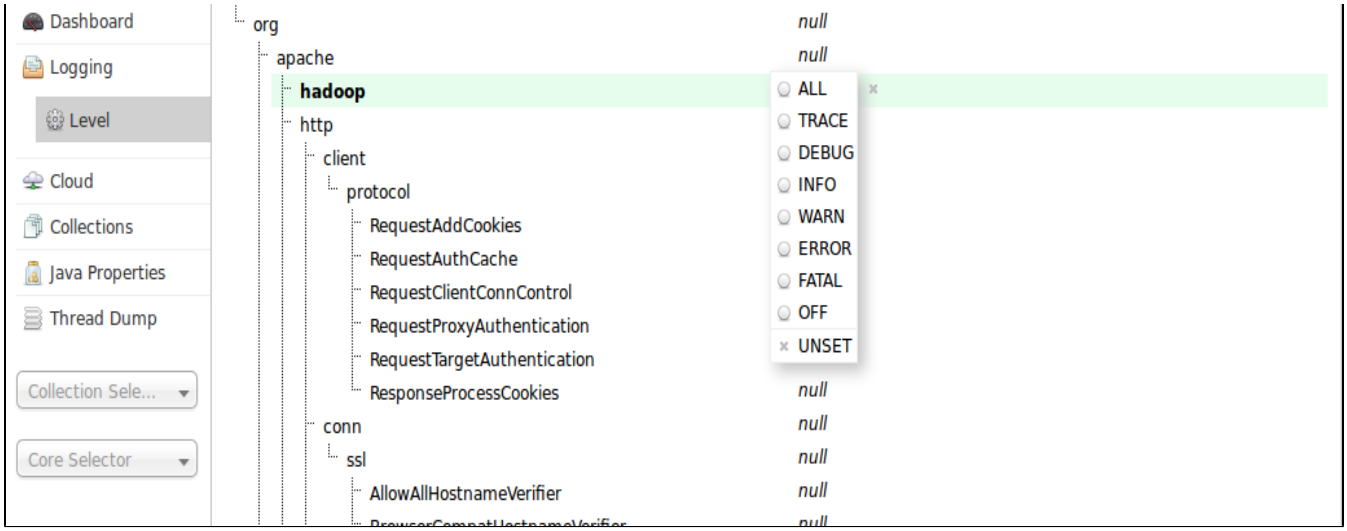


The Logging screen.

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:





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.
WARN	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

Solr uses Log4J version 1.2 for logging and is configured using `server/resources/log4j.properties`. Take a moment to inspect the contents of the `log4j.properties` file so that you are familiar with its structure. By default, Solr log messages will be written to `server/logs/solr.log` and to stdout (console), which is fine when you're just getting started with Solr.

When you're ready to deploy Solr in production, we recommend making a few minor changes to the log settings. Edit `log4j.properties` and set the `solr.log` property to the location where you want Solr to write log files, such as `/var/solr/logs`. Note that if you installed Solr as a service using the instructions provided at Taking Solr to Production, then see `/var/solr/log4j.properties` instead of the default `server/resources` vers ion.

```
solr.log=/var/solr/logs
```

Alternatively, you can use the `solr.solr.home` system property to set the location of the log files, such as:

```
solr.log=${solr.solr.home}/../logs
```

During initialization, Log4J will resolve this to a path based on the `solr.solr.home` system property. While you're customizing the `log4j.properties` file, we also recommend removing the CONSOLE appender from the rootLogger by changing the `log4j.rootLogger` property to:

```
log4j.rootLogger=INFO, file
```

Also, the default log rotation size threshold of 4MB is too small for production servers and should be increased to a larger value (such as 100MB or more).

```
log4j.appender.file.MaxFileSize=100MB
```

Logging Slow Queries

For high-volume search applications, logging every query can generate a large amount of logs and, depending on the volume, potentially impact performance. If you mine these logs for additional insights into your application, then logging every query request may be useful. On the other hand, if you're only concerned about warnings and error messages related to requests, then you can set the log verbosity to WARN. However, this poses a potential problem in that you won't know if any queries are slow, as slow queries are still logged at the INFO level. Solr provides a way to set your log verbosity threshold to WARN and be able to set a latency threshold above which a request is considered "slow" and log that request at the WARN level to help you identify slow queries in your application. To enable this behavior, configure the `<slowQueryThresholdMillis>` element in the **query** secti on of solrconfig.xml:

```
<slowQueryThresholdMillis>1000</slowQueryThresholdMillis>
```

Any queries that take longer than the specified threshold will be logged as "slow" queries at the WARN level.

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/8/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

Solr's `sample_techproducts_configs` config set uses the simple `<jmx />` configuration option. If you start the example with the necessary JVM system properties to launch an internal MBeanServer, Solr will register with it and you can connect using a tool like `jconsole`:

1. Launch the `techproducts` example with JMX enabled:

    ```
    bin/solr -e techproducts -Dcom.sun.management.jmxremote
    ```

2. Start `jconsole` (provided with the Sun JDK in the bin directory).
3. Connect to the "`start.jar`" shown in the list of local processes.
4. Switch to the "MBeans" tab. You should be able to see "`solr/techproducts`" listed there, at which point you can drill down and see details of every solr plugin.

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/8/docs/technotes/guides/management/agent.html



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.

MBean Request Handler

The MBean Request Handler offers programmatic access to the information provided on the Plugin/Stats page of the Admin UI.

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

The following examples assume you are running Solr's `techproducts` example configuration:

```
bin/solr start -e techproducts
```

To return information about the CACHE category only:

`http://localhost:8983/solr/techproducts/admin/mbeans?cat=CACHE`

To return information and statistics about the CACHE category only, formatted in JSON:

`http://localhost:8983/solr/techproducts/admin/mbeans?stats=true&cat=CACHE&indent=true&wt=json`

To return information for everything, and statistics for everything except the `fieldCache`:

`http://localhost:8983/solr/techproducts/admin/mbeans?stats=true&f.fieldCache.stats=false`

To return information and statistics for the `fieldCache` only:

`http://localhost:8983/solr/techproducts/admin/mbeans?key=fieldCache&stats=true`

Performance Statistics Reference

This page will explain all the JMX statistics that Solr exposes. The same stats are also exposed via the MBean Request Handler when stats are requested.

Request Handlers

Update Request Handler

The update request handler is an endpoint to send data to Solr. We can see how many update requests are being fired, how fast is it performing, and other valuable information regarding requests.

Path: /solr/<core>/update

Search Request Handler

Can be useful to measure and track number of search queries , response times etc. If you are not using the “select” handler then the path needs to be changed appropriately. Similarly if you are using the “sql” handler or “export” handler , the realtime handler “get”, or any other handler similar statistics can be found for that as well.

Path: /solr/<core>/select

Both Update Request Handler and Search Request Handler along with handlers like “sql”, “export” and realtime “get” handler will be having the following attributes in their statistics.

Attribute	Description
15minRateReqsPerSecond	Requests per second received over the past 15 minutes
5minRateReqsPerSecond	Requests per second received over the past 5 minutes
75thPcRequestTime	Request processing time for the request which belongs to the 75th Percentile. E.g. if 100 requests are received, then the 75th fastest request time will be reported by this stat.
95thPcRequestTime	Request processing time in milliseconds for the request which belongs to the 95th Percentile. E.g. if 80 requests are received, then the 76th fastest request time will be reported in this stat.
999thPcRequestTime	Request processing time in milliseconds for the request which belongs to the 99.9th Percentile. E.g. if 1000 requests are received, then the 999th fastest request time will be reported in this stat.
99thPcRequestTime	Request processing time in milliseconds for the request which belongs to the 99th Percentile. E.g. if 200 requests are received, then the 198th fastest request time will be reported in this stat.
avgRequestsPerSecond	Average number of requests received per second
avgTimePerRequest	Average time taken for processing the requests
errors	Number of error encountered by handler
clientErrors	Number of syntax errors/parse errors made by client while making requests.

handlerStart	Epoch time when the handler was registered.
medianRequestTime	Median of all the request processing time
requests	Total number of requests made since the Solr process was started
serverErrors	Number of errors thrown by the server while executing the request
timeouts	Number of responses received with partial results
totalTime	The sum of all request processing times since the Solr process was started

Update Handler

Update Handler: This section has information on the total number of adds, how many commits have been fired against a solr core

Path: /solr/<core>/updateHandler/DirectUpdateHandler2

Attribute	Description
adds	Total number of "add" requests since last commit
autocommit maxTime	Maximum time between two autocommits execution
autocommits	Total number of auto-commits executed
cumulative_adds	Number of "effective" additions executed over the lifetime. The counter is incremented when "add' command is executed while decremented when "rollback" is executed.
cumulative_deletesById	Number of document deletions executed by ID over the lifetime. The counter is incremented when "delete" command is executed while decremented when "rollback" is executed.
cumulative_deletesByQuery	Number of document deletions executed by query over the lifetime. The counter is incremented when "delete" command is executed while decremented when "rollback" is executed.
cumulative_errors	Number of error messages received by Update Handler while performing addition/deletion action on documents over the lifetime
deletesById	Currently uncommitted deletions by ID
deletesByQuery	Currently uncommitted deletions by query
docsPending	Number of documents which are pending commit
errors	Number of error messages received by Update Handler while performing addition/deletion/commit/rollback action on documents over the lifetime
expungeDeletes	Number of commit commands issued with expunge deletes
optimizes	Number of explicit optimize commands issued
rollbacks	Number of rollbacks executed

soft autocommit maxTime	Maximum documents ‘adds’ between two soft auto-commits.
soft autocommits	Number of soft commits executed
transaction_logs_total_number	Number of TLogs created from the beginning of the Solr instance. It will be equivalent to number of Hard commits executed.
transaction_logs_total_size	Total size of all the TLogs created so far from the beginning of the Solr instance

Caches

Document Cache

This cache holds Lucene Document objects (the stored fields for each document). Since Lucene internal document IDs are transient, this cache cannot be auto-warmed.

Path: /solr/<cache>/documentCache

Query Result Cache

This cache holds the results of previous searches: ordered lists of document IDs based on a query, a sort, and the range of documents requested

Path: /solr/<cache>/queryResultCache

Filter Cache

This cache is used for filters for unordered sets of all documents that match a query.

Path: /solr/<cache>/filterCache

Attribute	Description
cumulative_evictions	Number of cache evictions across all caches since this node has been running.
cumulative_hitratio	Ratio of cache hits to lookups across all the caches since this node has been running.
cumulative_hits	Number of cache hits across all the caches since this node has been running
cumulative_inserts	Number of cache insertions across all the caches since this node has been running
cumulative_lookups	Number of cache lookups across all the caches since this node has been running
evictions	Number of cache evictions for the current index searcher
hitratio	Ratio of cache hits to lookups for the current index searcher
hits	Number of hits for the current index searcher
inserts	Number of inserts into the cache

lookups	Number of lookups against the cache
size	Size of the cache at that particular instance (in KBs)
warmupTime	Warm-up time for the registered index searcher. This time is taken in account for the “auto-warming” of caches.

More on Solr caches

These statistics are per core. When you are running SolrCloud these statistics would co-relate to each performance of an individual replica

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. Queries and updates can be sent to any server. Solr will use the information in the ZooKeeper database to figure out which servers need to handle the request.

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
- SolrCloud Configuration and Parameters
  - Setting Up an External ZooKeeper Ensemble
  - Using ZooKeeper to Manage Configuration Files
  - ZooKeeper Access Control
  - Collections API
  - Parameter Reference
  - Command Line Utilities
  - SolrCloud with Legacy Configuration Files
  - ConfigSets API
- Rule-based Replica Placement
- Cross Data Center Replication (CDCR)

Getting Started with SolrCloud

SolrCloud is designed to provide a highly available, fault tolerant environment for distributing your indexed content and query requests across multiple servers. It's a system in which data is organized into multiple pieces, or shards, that can be hosted 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 to start Solr in SolrCloud mode, so you can begin to get a sense for how shards interact with each other during indexing and when serving queries. To that end, 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 start a SolrCloud cluster using startup scripts and a specific configset.

This tutorial assumes that you're already familiar with the basics of using Solr. If you need a refresher, please see 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.

SolrCloud Example

Interactive Startup

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. These nodes will each exist on a single machine, but will use different ports to mimic operation on different servers.

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 show you the command it uses to start the server, such as:

```
solr start -cloud -s example/cloud/node1/solr -p 8983
```

The first node will also start an embedded ZooKeeper server bound to port 9983. The Solr home for the first node is in `example/cloud/node1/solr` as indicated by the `-s` option.

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 might want to choose a name more appropriate 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 on, 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 **basic_configs**, **data_driven_schema_configs**, or **sample_techproducts_configs**. The configuration directories are pulled from `server/solr/configsets/` so you can review them beforehand if you wish. The **data_drive**

n_schema_configs configuration (the default) 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 status command:

```
$ bin/solr status
```

If you encounter any errors during this process, check the Solr log files in `example/cloud/node1/logs` and `example/cloud/node2/logs`.

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. Solr also provides a way to perform basic diagnostics for a collection using the healthcheck command:

```
$ bin/solr healthcheck -c gettingstarted
```

The healthcheck command gathers basic information about each replica in a collection, such as number of docs, current status (active, down, etc), and address (where the replica lives in the cluster).

Documents can now be added to SolrCloud using the Post Tool.

To stop Solr in SolrCloud mode, you would use the `bin/solr` script and issue the `stop` command, as in:

```
$ bin/solr stop -all
```

Starting with -noprompt

You can also get SolrCloud started with all the defaults instead of the interactive session using the following command:

```
$ bin/solr -e cloud -noprompt
```

Restarting Nodes

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 -s example/cloud/node1/solr
```

To restart node2 running on port 7574, you can do:

```
$ bin/solr restart -c -p 7574 -z localhost:9983 -s example/cloud/node2/solr
```

Notice that you need to specify the ZooKeeper address (-z localhost:9983) when starting node2 so that it can join the cluster with node1.

Adding a node to a cluster

Adding a node to an existing cluster is a bit advanced and involves a little more understanding of Solr. Once you startup a SolrCloud cluster using the startup scripts, you can add a new node to it by:

```
$ mkdir <solr.home for new solr node>
$ cp <existing solr.xml path> <new solr.home>
$ bin/solr start -cloud -s solr.home/solr -p <port num> -z <zk hosts string>
```

Notice that the above requires you to create a Solr home directory. You either need to copy `solr.xml` to the `solr_home` directory, or keep in centrally in ZooKeeper `/solr.xml`.

Example (with directory structure) that adds a node to an example started with "bin/solr -e cloud":

```
$ mkdir -p example/cloud/node3/solr
$ cp server/solr/solr.xml example/cloud/node3/solr
$ bin/solr start -cloud -s example/cloud/node3/solr -p 8987 -z localhost:9983
```

The previous command will start another Solr node on port 8987 with Solr home set to `example/cloud/node3/solr`. The new node will write its log files to `example/cloud/node3/logs`.

Once you're comfortable with how the SolrCloud example works, we recommend using the process described in Taking Solr to Production for setting up SolrCloud nodes in production.

How SolrCloud Works

The following sections cover provide general information about how various SolrCloud features work. To understand these features, it's important to first understand a few key concepts that relate to SolrCloud.

- Shards and Indexing Data in SolrCloud
- Distributed Requests
- Read and Write Side Fault Tolerance

If you are already familiar with SolrCloud concepts and basic functionality, you can skip to the section covering SolrCloud Configuration and Parameters.

Key SolrCloud Concepts

A SolrCloud cluster consists of some "logical" concepts layered on top of some "physical" concepts.

Logical

- A Cluster can host multiple Collections of Solr Documents.
- A collection can be partitioned into multiple Shards, which contain a subset of the Documents in the Collection.
- The number of Shards that a Collection has determines:
  - The theoretical limit to the number of Documents that Collection can reasonably contain.
  - The amount of parallelization that is possible for an individual search request.

Physical

- A Cluster is made up of one or more Solr Nodes, which are running instances of the Solr server process.
- Each Node can host multiple Cores.
- Each Core in a Cluster is a physical Replica for a logical Shard.
- Every Replica uses the same configuration specified for the Collection that it is a part of.
- The number of Replicas that each Shard has determines:
  - The level of redundancy built into the Collection and how fault tolerant the Cluster can be in the event that some Nodes become unavailable.

- The theoretical limit in the number concurrent search requests that can be processed under heavy load.

Shards and Indexing Data in SolrCloud

When your collection is too large for one node, you can break it up and store it in sections by creating multiple **shards**.

A Shard is a logical partition of the collection, containing a subset of documents from the collection, such that every document in a collection is contained in exactly one Shard. Which shard contains a each document in a collection depends on the overall "Sharding" strategy for that collection. For example, you might have a collection where the "country" field of each document determines which shard it is part of, so documents from the same country are co-located. A different collection might simply use a "hash" on the uniqueKey of each document to determine it's Shard.

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 an index across shards is not exclusively a SolrCloud concept. There were, however, several problems with the distributed approach that necessitated improvement with SolrCloud:

1. Splitting an index 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.

In SolrCloud there are no masters or slaves. Instead, every shard consists of at least one physical **replica**, exactly one of which is a **leader**. 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 the other replicas is automatically elected as the new leader.

When a document is sent to a Solr node for indexing, the system first determines which Shard that document belongs to, and then which node is currently hosting the leader for that shard. The document is then forwarded to the current leader for indexing, and the leader forwards the update to all of the other 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 (default) "`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"

Another use case could be if the customer "IBM" has a lot of documents and you want to spread it across multiple shards. The syntax for such a use case would be : "shard_key/num!document_id" where the /num is the number of bits from the shard key to use in the composite hash.

So "IBM/3!12345" will take 3 bits from the shard key and 29 bits from the unique doc id, spreading the tenant over 1/8th of the shards in the collection. Likewise if the num value was 2 it would spread the documents across 1/4th the number of shards. At query time, you include the prefix(es) along with the number of bits into your query with the `_route_` parameter (i.e., `q=solr&_route_=IBM/3!`) to direct queries to specific shards.

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

When you create a collection in SolrCloud, you decide on the initial number shards to be used. But 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.

Ignoring Commits from Client Applications in SolrCloud

In most cases, when running in SolrCloud mode, indexing client applications should not send explicit commit requests. Rather, you should configure auto commits with `openSearcher=false` and auto soft-commits to make recent updates visible in search requests. This ensures that auto commits occur on a regular schedule in the cluster. To enforce a policy where client applications should not send explicit commits, you should update all client applications that index data into SolrCloud. However, that is not always feasible, so Solr provides the IgnoreCommitOptimizeUpdateProcessorFactory, which allows you to ignore explicit commits and/or optimize requests from client applications without having refactor your client application code. To activate this request processor you'll need to add the following to your solrconfig.xml:

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <int name="statusCode">200</int>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

As shown in the example above, the processor will return 200 to the client but will ignore the commit / optimize request. Notice that you need to wire-in the implicit processors needed by SolrCloud as well, since this custom

chain is taking the place of the default chain.

In the following example, the processor will raise an exception with a 403 code with a customized error message:

```
<updateRequestProcessorChain name="ignore-commit-from-client" default="true">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <int name="statusCode">403</int>
    <str name="responseMessage">Thou shall not issue a commit!</str>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory" />
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

Lastly, you can also configure it to just ignore optimize and let commits pass thru by doing:

```
<updateRequestProcessorChain name="ignore-optimize-only-from-client-403">
  <processor class="solr.IgnoreCommitOptimizeUpdateProcessorFactory">
    <str name="responseMessage">Thou shall not issue an optimize, but commits are
OK!</str>
    <bool name="ignoreOptimizeOnly">true</bool>
  </processor>
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

Distributed Requests

When a Solr node receives a search request, that request is routed behinds the scenes to a replica of some shard that is part of the collection being searched. The chosen replica will act as an aggregator: creating internal requests to randomly chosen replicas of every shard in the collection, coordinating the responses, issuing any subsequent internal requests as needed (For example, to refine facets values, or request additional stored fields) and constructing the final response for the client.

Limiting Which Shards are Queried

While one of the advantages of using SolrCloud is the ability to query very large collections distributed among various shards, in some cases you may know that you are only interested in results from a subset of your shards. 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/gettingstarted/select?q=*:*
```

If, on the other hand, you wanted to search just one shard, you can specify that shard by it's logical ID, as in:

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1
```

If you want to search a group of shard Ids, you can specify them together:

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1,shard2
```

In both of the above examples, the shard Id(s) will be used to pick a random replica of that shard.

Alternatively, you can specify the explict replicas you wish to use in place of a shard Ids:

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=localhost:7574/solr/ge
ttingstarted,localhost:8983/solr/gettingstarted
```

Or you can specify a list of replicas to choose from for a single shard (for load balancing purposes) by using the pipe symbol (|):

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=localhost:7574/solr/ge
ttingstarted|localhost:7500/solr/gettingstarted
```

And of course, you can specify a list of shards (seperated by commas) each defined by a list of replicas (seperated by pipes). In this example, 2 shards are queried, the first being a random replica from shard1, the second being a random replica from the explicit pipe delimited list:

```
http://localhost:8983/solr/gettingstarted/select?q=*:*&shards=shard1,localhost:7574/
solr/gettingstarted|localhost:7500/solr/gettingstarted
```

Configuring the ShardHandlerFactory

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 in the solrconfig.xml:

```
<requestHandler name="standard" class="solr.SearchHandler" default="true">
  <!-- other params go here -->
  <shardHandler 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 concurrent connections that is made to each individual shard in a distributed search.
`maxConnections`	`10000`	The total maximum number of concurrent connections in distributed searches.
`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.

Configuring statsCache (Distributed IDF)

Document and term statistics are needed in order to calculate relevancy. Solr provides four implementations out of the box when it comes to document stats calculation:

- `LocalStatsCache`: This only uses local term and document statistics to compute relevance. In cases with uniform term distribution across shards, this works reasonably well.
  This option is the default if no `<statsCache>` is configured.
- `ExactStatsCache`: This implementation uses global values (across the collection) for document frequency.
- `ExactSharedStatsCache`: This is exactly like the exact stats cache in it's functionality but the global stats are reused for subsequent requests with the same terms.
- `LRUStatsCache`: This implementation uses an LRU cache to hold global stats, which are shared between requests.

The implementation can be selected by setting `<statsCache>` in `solrconfig.xml`. For example, the following line makes Solr use the `ExactStatsCache` implementation:

```
<statsCache class="org.apache.solr.search.stats.ExactStatsCache"/>
```

Avoiding Distributed Deadlock

Each shard serves top-level query requests and then makes sub-requests to all of the other shards. Care should be taken to ensure that the max number of threads serving HTTP requests 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 incoming requests will be blocked until the other pending requests are finished, but they will not finish since they are waiting for the sub-requests. By ensuring that Solr is configured to handle a sufficient number of threads, you can avoid deadlock situations like this.

Prefer Local Shards

Solr allows you to pass an optional boolean parameter named `preferLocalShards` to indicate that a distributed query should prefer local replicas of a shard when available. In other words, if a query includes `preferLocalShards=true`, then the query controller will look for local replicas to service the query instead of selecting replicas at random from across the cluster. This is useful when a query requests many fields or large fields to be returned per document because it avoids moving large amounts of data over the network when it is available locally. In addition, this feature can be useful for minimizing the impact of a problematic replica with degraded performance, as it reduces the likelihood that the degraded replica will be hit by other healthy replicas.

Lastly, it follows that the value of this feature diminishes as the number of shards in a collection increases because the query controller will have to direct the query to non-local replicas for most of the shards. In other words, this feature is mostly useful for optimizing queries directed towards collections with a small number of shards and many replicas. Also, this option should only be used if you are load balancing requests across all nodes that host replicas for the collection you are querying, as Solr's CloudSolrClient will do. If not load-balancing, this feature can introduce a hotspot in the cluster since queries won't be evenly distributed across the cluster.

Read and Write Side Fault Tolerance

SolrCloud supports elasticity, high availability, and fault tolerance in reads and writes. What this means, basically, is that when you have a large cluster, you can always make requests to the cluster: Reads will return results whenever possible, even if some nodes are down, and Writes will be acknowledged only if they are durable; i.e., you won't lose data.

Read Side Fault Tolerance

In a SolrCloud cluster each individual node load balances read requests across all the replicas in collection. You still need a load balancer on the 'outside' that talks to the cluster, or you need a smart client which understands how to read and interact with Solr's metadata in ZooKeeper and only requests the ZooKeeper ensemble's address to start discovering to which nodes it should send requests. (Solr provides a smart Java SolrJ client called CloudSolrClient.)

Even if some nodes in the cluster are offline or unreachable, a Solr node will be able to correctly respond to a search request as long as it can communicate with at least one replica of every shard, or one replica of every *relevant* shard if the user limited the search via the '`shards`' or '`_route_`' parameters. The more replicas there are of every shard, the more likely that the Solr cluster will be able to handle search results in the event of node failures.

`zkConnected`

A Solr node will return the results of a search request as long as it can communicate with at least one replica of every shard that it knows about, even if it can *not* communicate with ZooKeeper at the time it receives the request. This is normally the preferred behavior from a fault tolerance standpoint, but may result in stale or incorrect results if there have been major changes to the collection structure that the node has not been informed of via ZooKeeper (ie: shards may have been added or removed, or split into sub-shards)

A `zkConnected` header is included in every search response indicating if the node that processed the request was connected with ZooKeeper at the time:

Solr Response with partialResults

```
{
  "responseHeader": {
    "status": 0,
    "zkConnected": true,
    "QTime": 20,
    "params": {
      "q": "*:*"
    }
  },
  "response": {
    "numFound": 107,
    "start": 0,
    "docs": [ ... ]
  }
}
```

shards.tolerant

In the event that one or more shards queried are completely unavailable, then Solr's default behavior 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,
    "zkConnected": true,
    "partialResults": true,
    "QTime": 20,
    "params": {
      "q": "*:*"
    }
  },
  "response": {
    "numFound": 77,
    "start": 0,
    "docs": [ ... ]
  }
}
```

Write Side Fault Tolerance

SolrCloud is designed to replicate documents to ensure redundancy for your data, and enable you to send update requests to any node in the cluster. That node will determine if it hosts the leader for the appropriate shard, and if not it will forward the request to the the leader, which will then forwards it to all existing replicas, using versioning to make sure every replica has the most up-to-date version. If the leader goes down, and other

replica can take it's place. 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.

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 a replica is too far out of sync, 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.

SolrCloud Configuration and Parameters

In this section, we'll cover the various configuration options for SolrCloud.

The following sections cover these topics:

- Setting Up an External ZooKeeper Ensemble
- Using ZooKeeper to Manage Configuration Files
- ZooKeeper Access Control
- Collections API
- Parameter Reference
- Command Line Utilities
- SolrCloud with Legacy Configuration Files
- ConfigSets API

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, as described below.

How Many ZooKeepers?
"For a ZooKeeper service to be active, there must be a majority of non-failing machines that can communicate with each other. ***To create a deployment that can tolerate the failure of F machines, you should count on deploying 2xF+1 machines***. Thus, a deployment that consists of three machines can handle one failure, and a deployment of five machines can handle two failures. Note that a deployment of six machines can only handle two failures since three machines is not a majority.

For this reason, ZooKeeper deployments are usually made up of an odd number of machines."

-- *ZooKeeper Administrator's Guide*.

When planning how many ZooKeeper nodes to configure, keep in mind that the main principle for a ZooKeeper ensemble is maintaining a majority of servers to serve requests. This majority is also called a *quorum*. It is generally recommended to have an odd number of ZooKeeper servers in your ensemble, so a majority is maintained. For example, if you only have two ZooKeeper nodes and one goes down, 50% of available servers is not a majority, so ZooKeeper will no longer serve requests. However, if you have three ZooKeeper nodes and one goes down, you have 66% of available servers available, and ZooKeeper will continue normally while you repair the one down node. If you have 5 nodes, you could continue operating with two down nodes if necessary. More information on ZooKeeper clusters is available from the ZooKeeper documentation at http://zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html#sc_zkMulitServerSetup.

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 currently 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 `-z` parameter when using the bin/solr script. For example, in order to point the Solr instance to the ZooKeeper you've started on port 2181, this is what you'd need to do:

Starting `cloud` example with Zookeeper already running at port 2181 (with all other defaults):

```
bin/solr start -e cloud -z localhost:2181 -noprompt
```

Add a node pointing to an existing ZooKeeper at port 2181:

```
bin/solr start -cloud -s <path to solr home for new node> -p 8987 -z localhost:2181
```

NOTE: When you are not using an example to start solr, make sure you upload the configuration set to zookeeper before creating the collection.

Shut down ZooKeeper

To shut down ZooKeeper, use the zkServer script with the "stop" command: `zkServer.sh stop`.

Setting up a ZooKeeper Ensemble

With an external ZooKeeper ensemble, you need to set things up just a little more carefully as compared to the Getting Started example.

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: Amount of time, in ticks, to allow followers to connect and sync to a 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 and sync with the leader.

syncLimit: Amount of time, in ticks, to allow followers to sync with ZooKeeper. If followers fall too far behind a leader, they will be dropped.

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:

```
bin/solr start -e cloud -z localhost:2181,localhost:2182,localhost:2183 -noprompt
```

For more information on getting the most power from your ZooKeeper installation, check out the ZooKeeper Administrator's Guide.

Securing the ZooKeeper connection

You may also want to secure the communication between ZooKeeper and Solr.

To setup ACL protection of znodes, see ZooKeeper Access Control.

Using ZooKeeper to Manage Configuration Files

With SolrCloud your configuration files are kept in ZooKeeper. These files are uploaded in either of the following cases:

- When you start a SolrCloud example using the `bin/solr` script.
- When you create a collection using the `bin/solr` script.
- Explicitly upload a configuration set to ZooKeeper.

Startup Bootstrap

When you try SolrCloud for the first time using the `bin/solr -e cloud`, the related configset gets uploaded to zookeeper automatically and is linked with the newly created collection.

The below command would start SolrCloud with the default collection name (gettingstarted) and default configset (data_driven_schema_configs) uploaded and linked to it.

```
$ bin/solr -e cloud -noprompt
```

You can also explicitly upload a configuration directory when creating a collection using the `bin/solr` script with the `-d` option, such as:

```
$ bin/solr create -c mycollection -d data_driven_schema_configs
```

The create command will upload a copy of the `data_driven_schema_configs` configuration directory to ZooKeeper under `/configs/mycollection`. Refer to the Solr Start Script Reference page for more details about the create command for creating collections.

Once a configuration directory has been uploaded to ZooKeeper, you can update them using the ZooKeeper Command Line Interface (zkCLI).


It's a good idea to keep these files under version control.

Uploading configs using zkcli or SolrJ

In production situations, Config Sets can also be uploaded to ZooKeeper independent of collection creation using either Solr's zkcli.sh script, or the CloudSolrClient.uploadConfig java method.

The below command can be used to upload a new configset using the zkcli script.

```
$  sh zkcli.sh -cmd upconfig -zkhost  <host:port> -confname <name for configset>
-solrhome <solrhome> -confdir <path to directory with configset>
```

More information about the ZooKeeper Command Line Utility to help manage changes to configuration files, can be found in the section on Command Line Utilities.

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.

Preparing ZooKeeper before first cluster start

If you will share the same ZooKeeper instance with other applications you should use a *chroot* in ZooKeeper. Please see Taking Solr to Production#ZooKeeperchroot for instructions.

There are certain configuration files containing cluster wide configuration. Since some of these are crucial for the cluster to function properly, you may need to upload such files to ZooKeeper before starting your Solr cluster for the first time. Examples of such configuration files (not exhaustive) are `solr.xml`, `security.json` and `clusterprops.json`.

If you for example would like to keep your `solr.xml` in ZooKeeper to avoid having to copy it to every node's `solr_home` directory, you can push it to ZooKeeper with the `zkcli.sh` utility (Unix example):

```
zkcli.sh -zkhost localhost:2181 -cmd putfile /solr.xml /path/to/solr.xml
```

ZooKeeper Access Control

This section describes using ZooKeeper access control lists (ACLs) with Solr. For information about ZooKeeper ACLs, see the ZooKeeper documentation at http://zookeeper.apache.org/doc/r3.4.6/zookeeperProgrammers.html#sc_ZooKeeperAccessControl.

- About ZooKeeper ACLs
- How to Enable ACLs
- Changing ACL Schemes

About ZooKeeper ACLs

SolrCloud uses ZooKeeper for shared information and for coordination.

This section describes how to configure Solr to add more restrictive ACLs to the ZooKeeper content it creates, and how to tell Solr about the credentials required to access the content in ZooKeeper. If you want to use ACLs in your ZooKeeper nodes, you will have to activate this functionality; by default, Solr behavior is open-unsafe ACL everywhere and uses no credentials.

Content stored in ZooKeeper is critical to the operation of a SolrCloud cluster. Open access to SolrCloud content on ZooKeeper could lead to a variety of problems. For example:

- Changing configuration might cause Solr to fail or behave in an unintended way.
- Changing cluster state information into something wrong or inconsistent might very well make a SolrCloud cluster behave strangely.
- Adding a delete-collection job to be carried out by the Overseer will cause data to be deleted from the cluster.

You may want to enable ZooKeeper ACLs with Solr if you grant access to your ZooKeeper ensemble to entities you do not trust, or if you want to reduce risk of bad actions resulting from, e.g.:

- Malware that found its way into your system.
- Other systems using the same ZooKeeper ensemble (a "bad thing" might be done by accident).

You might even want to limit read-access, if you think there is stuff in ZooKeeper that not everyone should know about. Or you might just in general work on a need-to-know basis.

Protecting ZooKeeper itself could mean many different things. **This section is about protecting Solr content in ZooKeeper**. ZooKeeper content basically lives persisted on disk and (partly) in memory of the ZooKeeper processes. **This section is not about protecting ZooKeeper data at storage or ZooKeeper process levels** - that's for ZooKeeper to deal with.

But this content is also available to "the outside" via the ZooKeeper API. Outside processes can connect to ZooKeeper and create/update/delete/read content; for example, a Solr node in a SolrCloud cluster wants to create/update/delete/read, and a SolrJ client wants to read from the cluster. It is the responsibility of the outside processes that create/update content to setup ACLs on the content. ACLs describe who is allowed to read, update, delete, create, etc. Each piece of information (znode/content) in ZooKeeper has its own set of ACLs, and inheritance or sharing is not possible. The default behavior in Solr is to add one ACL on all the content it creates - one ACL that gives anyone the permission to do anything (in ZooKeeper terms this is called "the open-unsafe ACL").

How to Enable ACLs

We want to be able to:

1. Control the credentials Solr uses for its ZooKeeper connections. The credentials are used to get permission to perform operations in ZooKeeper.
2. Control which ACLs Solr will add to znodes (ZooKeeper files/folders) it creates in ZooKeeper.
3. Control it "from the outside", so that you do not have to modify and/or recompile Solr code to turn this on.

Solr nodes, clients and tools (e.g. ZkCLI) always use a java class called `SolrZkClient` to deal with their ZooKeeper stuff. The implementation of the solution described here is all about changing `SolrZkClient`. If you use `SolrZkClient` in your application, the descriptions below will be true for your application too.

Controlling Credentials

You control which credentials provider will be used by configuring the `zkCredentialsProvider` property in `solr.xml`'s `<solrcloud>` section to the name of a class (on the classpath) implementing the `ZkCredentialsProvider` interface. `server/solr/solr.xml` in the Solr distribution defines the `zkCredentialsProvider` such that it will take on the value of the same-named `zkCredentialsProvider` system property if it is defined (e.g. by uncommenting the `SOLR_ZK_CREDS_AND_ACLS` environment variable definition in `solr.in.sh/.cmd` - see below) , or if not, default to the `DefaultZkCredentialsProvider` implementation.

Out of the Box Implementations

You can always make you own implementation, but Solr comes with two implementations:

- `org.apache.solr.common.cloud.DefaultZkCredentialsProvider`: Its `getCredentials()` returns a list of length zero, or "no credentials used". This is the default.
- `org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDigestZkCredentialsProvider`: This lets you define your credentials using system properties. It supports at most one set of credentials.
  - The schema is "digest". The username and password are defined by system properties "`zkDigestUsername`" and "`zkDigestPassword`", respectively. This set of credentials will be added to the list of credentials returned by `getCredentials()` if both username and password are provided.
  - If the one set of credentials above is not added to the list, this implementation will fall back to default behavior and use the (empty) credentials list from `DefaultZkCredentialsProvider`.

Controlling ACLs

You control which ACLs will be added by configuring `zkACLProvider` property in `solr.xml`'s `<solrcloud>` section to the name of a class (on the classpath) implementing the `ZkACLProvider` interface. `server/solr/solr.xml` in the Solr distribution defines the `zkACLProvider` such that it will take on the value of the same-named `zkACLProvider` system property if it is defined (e.g. by uncommenting the `SOLR_ZK_CREDS_AND_ACLS` environment variable definition in `solr.in.sh/.cmd` - see below) , or if not, default to the `DefaultZkACLProvider` implementation.

Out of the Box Implementations

You can always make you own implementation, but Solr comes with:

- `org.apache.solr.common.cloud.DefaultZkACLProvider:` It returns a list of length one for all `zNodePath`-s. The single ACL entry in the list is "open-unsafe". This is the default.
- `org.apache.solr.common.cloud.VMParamsAllAndReadonlyDigestZkACLProvider:` This lets you define your ACLs using system properties. Its `getACLsToAdd()` implementation does not use `zNodePath` for anything, so all znodes will get the same set of ACLs. It supports adding one or both of these options:
  - A user that is allowed to do everything.
    - The permission is "`ALL`" (corresponding to all of `CREATE`, `READ`, `WRITE`, `DELETE`, and `ADMIN`), and the schema is "digest".
    - The username and password are defined by system properties "`zkDigestUsername`" and "`zkDigestPassword`", respectively.
    - This ACL will not be added to the list of ACLs unless both username and password are provided.
  - A user that is only allowed to perform read operations.
    - The permission is "`READ`" and the schema is "digest".
    - The username and password are defined by system properties "`zkDigestReadonlyUsername`" and "`zkDigestReadonlyPassword`", respectively.
    - This ACL will not be added to the list of ACLs unless both username and password are provided.
- `org.apache.solr.common.cloud.SaslZkACLProvider:` Requires SASL authentication. Gives all permissions for the user specified in system property `solr.authorization.superuser` (default: `solr`) when using SASL, and gives read permissions for anyone else. Designed for a setup where configurations have already been set up and will not be modified, or where configuration changes are controlled via Solr APIs. This provider will be useful for administration in a kerberos environment. In such an environment, the administrator wants Solr to authenticate to ZooKeeper using SASL, since this is only way to authenticate with ZooKeeper via Kerberos.

If none of the above ACLs is added to the list, the (empty) ACL list of `DefaultZkACLProvider` will be used by default.

Notice the overlap in system property names with credentials provider `VMParamsSingleSetCredentialsDigestZkCredentialsProvider` (described above). This is to let the two providers collaborate in a nice and perhaps common way: we always protect access to content by limiting to two users - an admin-user and a readonly-user - AND we always connect with credentials corresponding to this same admin-user, basically so that we can do anything to the content/znodes we create ourselves.

You can give the readonly credentials to "clients" of your SolrCloud cluster - e.g. to be used by SolrJ clients. They will be able to read whatever is necessary to run a functioning SolrJ client, but they will not be able to modify any content in ZooKeeper.

bin/solr & solr.cmd, server/scripts/cloud-scripts/zkcli.sh & zkcli.bat

These Solr scripts can enable use of ZK ACLs by setting the appropriate system properties: uncomment the following and replace the passwords with ones you choose to enable the above-described VM parameters ACL and credentials providers in the following files:

solr.in.sh

```
# Settings for ZK ACL
#SOLR_ZK_CREDS_AND_ACLS="-DzkACLProvider=org.apache.solr.common.cloud.VMParamsAllAnd
ReadonlyDigestZkACLProvider \
#
-DzkCredentialsProvider=org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDig
estZkCredentialsProvider \
#  -DzkDigestUsername=admin-user -DzkDigestPassword=CHANGEME-ADMIN-PASSWORD \
#  -DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=CHANGEME-READONLY-PASSWORD"
#SOLR_OPTS="$SOLR_OPTS $SOLR_ZK_CREDS_AND_ACLS"
```

solr.in.cmd

```
REM Settings for ZK ACL
REM set
SOLR_ZK_CREDS_AND_ACLS=-DzkACLProvider=org.apache.solr.common.cloud.VMParamsAllAndRe
adonlyDigestZkACLProvider ^
REM
-DzkCredentialsProvider=org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDig
estZkCredentialsProvider ^
REM  -DzkDigestUsername=admin-user -DzkDigestPassword=CHANGEME-ADMIN-PASSWORD ^
REM  -DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=CHANGEME-READONLY-PASSWORD
REM set SOLR_OPTS=%SOLR_OPTS% %SOLR_ZK_CREDS_AND_ACLS%
```

zkcli.sh

```
# Settings for ZK ACL
#SOLR_ZK_CREDS_AND_ACLS="-DzkACLProvider=org.apache.solr.common.cloud.VMParamsAllAnd
ReadonlyDigestZkACLProvider \
#
-DzkCredentialsProvider=org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDig
estZkCredentialsProvider \
#  -DzkDigestUsername=admin-user -DzkDigestPassword=CHANGEME-ADMIN-PASSWORD \
#  -DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=CHANGEME-READONLY-PASSWORD"
```

zkcli.bat

```
REM Settings for ZK ACL
REM set
SOLR_ZK_CREDS_AND_ACLS=-DzkACLProvider=org.apache.solr.common.cloud.VMParamsAllAndRe
adonlyDigestZkACLProvider ^
REM
-DzkCredentialsProvider=org.apache.solr.common.cloud.VMParamsSingleSetCredentialsDig
estZkCredentialsProvider ^
REM  -DzkDigestUsername=admin-user -DzkDigestPassword=CHANGEME-ADMIN-PASSWORD ^
REM  -DzkDigestReadonlyUsername=readonly-user
-DzkDigestReadonlyPassword=CHANGEME-READONLY-PASSWORD
```

Changing ACL Schemes

Over the lifetime of operating your Solr cluster, you may decide to move from an unsecured ZooKeeper to a secured instance. Changing the configured `zkACLProvider` in `solr.xml` will ensure that newly created nodes are secure, but will not protect the already existing data. To modify all existing ACLs, you can use the `updateacls` command with Solr's ZkCLI. First uncomment the `SOLR_ZK_CREDS_AND_ACLS` environment variable definition in `server/scripts/cloud-scripts/zkcli.sh` (or `zkcli.bat` on Windows) and fill in the passwords for the admin-user and the readonly-user - see above - then run `server/scripts/cloud-scripts/zkcli.sh -cmd updateacls /zk-path`, or on Windows run `server\scripts\cloud-scripts\zkcli.bat cmd updateacls /zk-path` .

Changing ACLs in ZK should only be done while your SolrCloud cluster is stopped. Attempting to do so while Solr is running may result in inconsistent state and some nodes becoming inaccessible.

The VM properties `zkACLProvider` and `zkCredentialsProvider`, included in the `SOLR_ZK_CREDS_AND_ACLS` environment variable in `zkcli.sh/.bat`, control the conversion:

- The Credentials Provider must be one that has current admin privileges on the nodes. When omitted, the process will use no credentials (suitable for an unsecure configuration).
- The ACL Provider will be used to compute the new ACLs. When omitted, the process will set all permissions to all users, removing any security present.

The uncommented `SOLR_ZK_CREDS_AND_ACLS` environment variable in `zkcli.sh/.bat` sets the credentials and ACL providers to the `VMParamsSingleSetCredentialsDigestZkCredentialsProvider` and `VMParamsAllAndReadonlyDigestZkACLProvider` implementations, described earlier in the page.

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=MODIFYCOLLECTION`: Modify certain attributes of 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=DELETESTATUS`: Delete the stored response of a previous asynchronous request
`/admin/collections?action=LIST`: List all collections
`/admin/collections?action=ADDREPLICAPROP`: Add an arbitrary property to a replica specified by collection/shard/replica
`/admin/collections?action=DELETEREPLICAPROP`: Delete an arbitrary property from a replica specified by collection/shard/replica
`/admin/collections?action=BALANCESHARDUNIQUE`: Distribute an arbitrary property, one per shard, across the nodes in a collection
`/admin/collections?action=REBALANCELEADERS`: Distribute leader role based on the "preferredLeader" assignments
`/admin/collections?action=FORCELEADER`: Force a leader election in a shard if leader is lost
`/admin/collections?action=MIGRATESTATEFORMAT`: Migrate a collection from shared clusterstate.json to per-collection state.json
`/admin/collections?action=BACKUP`:Backup Solr indexes and configurations for a specifc collection
`/admin/collections?action=RESTORE`: Restore Solr indexes and configurations for a specific collection
`/admin/collections?action=DELETENODE`: Delete all replicas from a given node
`/admin/collections?action=REPLACENODE`: Move all replicas from a node to another node

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. Possible values are **implicit** or **compositeId**. The 'implicit' router does not automatically route documents to different shards. Whichever shard you indicate on the indexing request (or within each document) will be used as the destination for those documents. The 'compositeId' router hashes the value in the uniqueKey field and looks up that hash in the collection's clusterstate to determine which shard will receive the document, with the additional ability to manually direct the routing. 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		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`. Alternatively, use the special value of `EMPTY` to initially create no shard-replica within the new collection and then later use the ADDREPLICA operation to add shard-replica when and where required.

createNodeSet.shuffle	boolean	No	true	Controls wether or not the shard-replicas created for this collection will be assigned to the nodes specified by the createNodeSet in a sequential manner, or if the list of nodes should be shuffled prior to creating individual replicas. A 'false' value makes the results of a collection creation predictible and gives more exact control over the location of the individual shard-replicas, but 'true' can be a better choice for ensuring replicas are distributed evenly across nodes. Ignored if createNodeSet is not also specified.
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 the section Defining core.properties for details on supported properties and values.
autoAddReplicas	boolean	No	false	When set to true, enables auto addition of replicas on shared file systems. See the section autoAddReplicas Settings for more details on settings and overrides.
async	string	No		Request ID to track this action which will be processed asynchronously.
rule	string	No		Replica placement rules. See the section Rule-based Replica Placement for details.
snitch	string	No		Details of the snitch provider. See the section Rule-based Replica Placement for details.

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&numShards=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>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3597</int>
      </lst>
      <str name="core">newCollection_shard2_replica1</str>
    </lst>
  </lst>
</response>
```

Modify attributes of a Collection

`/admin/collections?action=MODIFYCOLLECTION&collection=<collection-name>&<attribute-name>=` *`<attribute-value>&<another-attribute-name>=<another-value>`*

It's possible to edit multiple attributes at a time. Changing these values only updates the z-node on Zookeeper, they do not change the topology of the collection. For instance, increasing replicationFactor will *not* automatically add more replicas to the collection but *will* allow more ADDREPLICA commands to succeed.

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection to be modified.
<attribute-name>	string	Yes	Key-value pairs of attribute names and attribute values. The attributes that can be modified are: • maxShardsPerNode • replicationFactor • autoAddReplicas • collection.configName • rule • snitch See the CREATE section above for details on these attributes.

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.
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 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 which are written to disk as two (new) shards. 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. A soft commit is automatically

issued after splitting a shard so that documents are made visible on sub-shards. An explicit commit (hard or soft) is not necessary after a split operation because the index is automatically persisted to disk during the split operation.

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, you should 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 the section Defining core.properties for details on supported properties and values.
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://localhost:8983/solr/admin/collections?action=SPLITSHARD&collection=anotherCollection&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>
    </lst>
    <lst>
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">3681</int>
      </lst>
      <str name="core">anotherCollection_shard1_0_replica1</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 the section Defining core.properties for details on supported properties and values.
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. 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://localhost:8983/solr/admin/collections?action=CREATESHARD&collection=anImplicitCollection&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, remove them from `clusterstate.json`, and (by default) delete the instanceDir and dataDir for each replica. 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.
deleteInstanceDir	boolean	No	By default Solr will delete the entire instanceDir of each replica that is deleted. Set this to `false` to prevent the instance directory from being deleted.
deleteDataDir	boolean	No	By default Solr will delete the dataDir of each replica that is deleted. Set this to `false` to prevent the data directory from being deleted.
deleteIndex	boolean	No	By default Solr will delete the index of each replica that is deleted. Set this to `false` to prevent the index directory from being deleted.
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. 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://localhost:8983/solr/admin/collections?action=DELETESHARD&collection=anotherCo
llection&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.
async	string	No	Request ID to track this action which will be processed asynchronously .

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://localhost:8983/solr/admin/collections?action=CREATEALIAS&name=testalias&colle
ctions=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.
async	string	No	Request ID to track this action which will be processed asynchronously .

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://localhost: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.
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 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://localhost: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>
    </lst>
    <lst name="10.0.1.4:8983_solr">
      <lst name="responseHeader">
        <int name="status">0</int>
        <int name="QTime">67</int>
      </lst>
    </lst>
  </lst>
</response>
```

Delete a Replica

`/admin/collections?action=DELETEREPLICA&collection=`*`collection`*`&shard=`*`shard`*`&replica=`*`replica`*

Delete a named replica from the specified collection and shard. If the corresponding core is up and running the core is unloaded, the entry is removed from the clusterstate, and (by default) delete the instanceDir and dataDir. 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.
deleteInstanceDir	boolean	No	By default Solr will delete the entire instanceDir of the replica that is deleted. Set this to `false` to prevent the instance directory from being deleted.
deleteDataDir	boolean	No	By default Solr will delete the dataDir of the replica that is deleted. Set this to `false` to prevent the data directory from being deleted.
deleteIndex	boolean	No	By default Solr will delete the index of the replica that is deleted. Set this to `false` to prevent the index directory from being deleted.
onlyIfDown	boolean	No	When set to 'true' will not take any action if the replica is active. Default 'false'

async	string	No	Request ID to track this action which will be processed asynchronously .

Examples

Input

```
http://localhost:8983/solr/admin/collections?action=DELETEREPLICA&collection=test2&s
hard=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	Yes*	The name of the shard to which replica is to be added. If shard is not specified, then _route_ must be.
route	string	No*	If the exact shard name is not known, users may pass the _route_ value and the system would identify the name of the shard. Ignored if the shard param is also specified.
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 Defining core.properties.
async	string	No	Request ID to track this action which will be processed asynchronously

Examples

Input

```
http://localhost:8983/solr/admin/collections?action=ADDREPLICA&collection=test2&shar
d=shard2&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>
    </lst>
  </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 supported properties names are `urlScheme` and `utoAddReplicas and location`. Other names are rejected with an error.
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=h
ttps
```

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 long running operation and therefore using the `async` parameter is highly recommended. If the async parameter is not specified then the operation is synchronous by default and keeping a large read timeout on the invocation is advised. Even with a large read timeout, 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 the section Defining core.properties for details on supported properties and values.
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=192.1
67.1.2:8983_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=19
2.167.1.2:8983_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",{
      ...
  }],
  "collection_operations":[
    "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.SolrExcepti
on: 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},
    ...
  ],
  "overseer_queue":[
    ...
  ],
  ...
```

Cluster Status

`/admin/collections?action=CLUSTERSTATUS`

Fetch the cluster status including collections, shards, replicas, configuration name 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.
route	string	No	This can be used if you need the details of the shard where a particular document belongs to and you don't know which shard it falls under.

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",
        "znodeVersion": 11,
        "autoCreated":"true",
  "configName" : "my_config",
        "aliases":["both_collections"]
      },
      "collection2":{
        ...
      }
    },
    "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 and response 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.

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>
```

Delete Status

`/admin/collections?action=DELETESTATUS&requestid=`*`request-id`*

Delete the stored response of an already failed or completed Asynchronous Collection API call.

Input

Query Parameters

Key	Type	Required	Description
requestid	string	No	The request-id of the async call we need to clear the stored response for.
flush	boolean	No	Set to true to clear all stored completed and failed async request responses.

Examples

Input: Valid Request Status

```
http://localhost:8983/solr/admin/collections?action=DELETESTATUS&requestid=foo
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <str name="status">successfully removed stored response for [foo]</str>
</response>
```

Input: Invalid RequestId

```
http://localhost:8983/solr/admin/collections?action=DELETESTATUS&requestid=bar
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <str name="status">[bar] not found in stored responses</str>
</response>
```

Input: Clearing up all the stored statuses

```
http://localhost:8983/solr/admin/collections?action=DELETESTATUS&flush=true
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <str name="status"> successfully cleared stored collection api responses </str>
</response>
```

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"]}
```

Add Replica Property

`/admin/collections?action=ADDREPLICAPROP&collection=collectionName&shard=shardName&replica=replicaName&property=propertyName&property.value=value`

Assign an arbitrary property to a particular replica and give it the value specified. If the property already exists, it will be overwritten with the new value.

Input

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection this replica belongs to.
shard	string	Yes	The name of the shard the replica belongs to.
replica	string	Yes	The replica, e.g. core_node1.

property (1)	string	Yes	The property to add. Note: this will have the literal 'property.' prepended to distinguish it from system-maintained properties. So these two forms are equivalent:
			`property=special`
			and
			`property=property.special`

property.value	string	Yes	The value to assign to the property.
shardUnique (1)	Boolean	No	default: false. If true, then setting this property in one replica will remove the property from all other replicas in that shard.

(1) There is one pre-defined property "preferredLeader" for which shardUnique is forced to 'true' and an error returned if shardUnique is explicitly set to 'false'. PreferredLeader is a boolean property, any value assigned that is not equal (case insensitive) to 'true' will be interpreted as 'false' for preferredLeader.

Output

Output Content

The response will include the status of the request. If the status is anything other than "0", an error message will explain why the request failed.

Examples

Input: This command would set the preferredLeader (property.preferredLeader) to true on core_node1, and remove that property from any other replica in the shard.

```
http://localhost:8983/solr/admin/collections?action=ADDREPLICAPROP&shard=shard1&coll
ection=collection1&replica=core_node1&property=preferredLeader&property.value=true
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">46</int>
  </lst>
</response>
```

Input: This pair of commands will set the "testprop" (property.testprop) to 'value1' and 'value2' respectively for two nodes in the same shard.

```
http://localhost:8983/solr/admin/collections?action=ADDREPLICAPROP&shard=shard1&coll
ection=collection1&replica=core_node1&property=testprop&property.value=value1

http://localhost:8983/solr/admin/collections?action=ADDREPLICAPROP&shard=shard1&coll
ection=collection1&replica=core_node3&property=property.testprop&property.value=valu
e2
```

Input: This pair of commands would result in core_node_3 having the testprop (property.testprop) value set because the second command specifies shardUnique=true, which would cause the property to be removed from core_node_1.

```
http://localhost:8983/solr/admin/collections?action=ADDREPLICAPROP&shard=shard1&coll
ection=collection1&replica=core_node1&property=testprop&property.value=value1

http://localhost:8983/solr/admin/collections?action=ADDREPLICAPROP&shard=shard1&coll
ection=collection1&replica=core_node3&property=testprop&property.value=value2&shardU
nique=true
```

Delete Replica Property

`/admin/collections?action=DELETEREPLICAPROP&collection=collectionName&shard=`*`shardName`*`&replica=`*`replicaName`*`&property=`*`propertyName`*

Deletes an arbitrary property from a particular replica.

Input

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection this replica belongs to
shard	string	Yes	The name of the shard the replica belongs to.
replica	string	Yes	The replica, e.g. core_node1.

property	string	Yes	The property to add. Note: this will have the literal 'property.' prepended to distinguish it from system-maintained properties. So these two forms are equivalent:
			`property=special`
			and
			`property=property.special`

Output

Output Content

The response will include the status of the request. If the status is anything other than "0", an error message will explain why the request failed.

Examples

Input: This command would delete the preferredLeader (property.preferredLeader) from core_node1.

```
http://localhost:8983/solr/admin/collections?action=DELETEREPLICAPROP&shard=shard1&c
ollection=collection1&replica=core_node1&property=preferredLeader
```

Output:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">9</int>
  </lst>
</response>
```

Balance a Property

`/admin/collections?action=BALANCESHARDUNIQUE&collection=`*`collectionName`*`&property=`*`propertyName`*

Insures that a particular property is distributed evenly amongst the physical nodes that make up a collection. If the property already exists on a replica, every effort is made to leave it there. If the property is *not* on any replica on a shard one is chosen and the property is added.

Input

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection to balance the property in.
property	string	Yes	The property to balance. The literal "property." is prepended to this property if not specified explicitly.
onlyactivenodes	boolean	No	Defaults to true. Normally, the property is instantiated on active nodes only. If this parameter is specified as "false", then inactive nodes are also included for distribution.
shardUnique	boolean	No	Something of a safety valve. There is one pre-defined property (preferredLeader) that defaults this value to "true". For all other properties that are balanced, this must be set to "true" or an error message is returned.

Output

Output Content

The response will include the status of the request. If the status is anything other than "0", an error message will explain why the request failed.

Examples

Input: Either of these commands would put the "preferredLeader" property on one replica in every shard in the "collection1" collection.

```
http://localhost:8983/solr/admin/collections?action=BALANCESHARDUNIQUE&collection=collection1&property=preferredLeader

http://localhost:8983/solr/admin/collections?action=BALANCESHARDUNIQUE&collection=collection1&property=property.preferredLeader
```

Output:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">9</int>
  </lst>
</response>
```

Examining the clusterstate after issuing this call should show exactly one replica in each shard that has this property.

Rebalance Leaders

Reassign leaders in a collection according to the preferredLeader property across active nodes.

`/admin/collections?action=REBALANCELEADERS&collection=collectionName`

Assigns leaders in a collection according to the preferredLeader property on active nodes. This command should be run after the preferredLeader property has been assigned via the BALANCESHARDUNIQUE or ADDREPLICAPROP commands. NOTE: it is not *required* that all shards in a collection have a preferredLeader property. Rebalancing will only attempt to reassign leadership to those replicas that have the preferredLeader property set to "true" *and* are not currently the shard leader *and* are currently active.

Input

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection to rebalance preferredLeaders on.
maxAtOnce	string	No	The maximum number of reassignments to have queue up at once. Values <=0 are use the default value Integer.MAX_VALUE. When this number is reached, the process waits for one or more leaders to be successfully assigned before adding more to the queue.
maxWaitSeconds	string	No	Defaults to 60. This is the timeout value when waiting for leaders to be reassigned. NOTE: if maxAtOnce is less than the number of reassignments that will take place, this is the maximum interval that any *single* wait for at least one reassignment. For example, if 10 reassignments are to take place and maxAtOnce is 1 and maxWaitSeconds is 60, the upper bound on the time that the command may wait is 10 minutes.

Output

Output Content

The response will include the status of the request. If the status is anything other than "0", an error message will explain why the request failed.

Examples

Input: Either of these commands would cause all the active replicas that had the "preferredLeader" property set and were *not* already the preferred leader to become leaders.

```
http://localhost:8983/solr/admin/collections?action=REBALANCELEADERS&collection=coll
ection1
http://localhost:8983/solr/admin/collections?action=REBALANCELEADERS&collection=coll
ection1&maxAtOnce=5&maxWaitSeconds=30
```

Output: In this example, two replicas in the "alreadyLeaders" section already had the leader assigned to the same node as the preferredLeader property so no action was taken. The replica in the "inactivePreferreds" section had the preferredLeader property set but the node was down and no action was taken. The three nodes in the "successes" section were made leaders because they had the preferredLeader property set but were not leaders and they were active.

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">123</int>
  </lst>
  <lst name="alreadyLeaders">
    <lst name="core_node1">
      <str name="status">success</str>
      <str name="msg">Already leader</str>
      <str name="nodeName">192.168.1.167:7400_solr</str>
    </lst>
    <lst name="core_node17">
      <str name="status">success</str>
      <str name="msg">Already leader</str>
      <str name="nodeName">192.168.1.167:7600_solr</str>
    </lst>
  </lst>
  <lst name="inactivePreferreds">
    <lst name="core_node4">
      <str name="status">skipped</str>
      <str name="msg">Node is a referredLeader, but it's inactive. Skipping</str>
      <str name="nodeName">192.168.1.167:7500_solr</str>
    </lst>
  </lst>
  <lst name="successes">
    <lst name="_collection1_shard3_replica1">
      <str name="status">success</str>
      <str name="msg">
        Assigned 'Collection: 'collection1', Shard: 'shard3', Core:
'collection1_shard3_replica1', BaseUrl:
        'http://192.168.1.167:8983/solr'' to be leader
      </str>
    </lst>
    <lst name="_collection1_shard5_replica3">
      <str name="status">success</str>
      <str name="msg">
        Assigned 'Collection: 'collection1', Shard: 'shard5', Core:
'collection1_shard5_replica3', BaseUrl:
        'http://192.168.1.167:7200/solr'' to be leader
      </str>
    </lst>
    <lst name="_collection1_shard4_replica2">
      <str name="status">success</str>
      <str name="msg">
        Assigned 'Collection: 'collection1', Shard: 'shard4', Core:
'collection1_shard4_replica2', BaseUrl:
        'http://192.168.1.167:7300/solr'' to be leader
      </str>
    </lst>
  </lst>
</response>
```

Examining the clusterstate after issuing this call should show that every live node that has the "preferredLeader" property should also have the "leader" property set to *true*.

Force Shard Leader

In the unlikely event of a shard losing its leader, this command can be invoked to force the election of a new leader

`/admin/collections?action=FORCELEADER&collection=<collectionName>&shard=<shardName>`

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The name of the collection
shard	string	Yes	The name of the shard

This is an expert level command, and should be invoked only when regular leader election is not working. This may potentially lead to loss of data in the event that the new leader doesn't have certain updates, possibly recent ones, which were acknowledged by the old leader before going down.

Migrate Cluster State

A Expert level utility API to move a collection from shared `clusterstate.json` zookeeper node (created with `stateFormat=1`, the default in all Solr releases prior to 5.0) to the per-collection `state.json` stored in ZooKeeper (created with `stateFormat=2`, the current default) seamlessly without any application down-time.

`/admin/collections?action=MIGRATESTATEFORMAT&collection=<collection_name>`

Key	Type	Required	Description
collection	string	Yes	The name of the collection to be migrated from `clusterstate.json` to its own `state.json` zookeeper node
async	string	No	Request ID to track this action which will be processed asynchronously .

This API is useful in migrating any collections created prior to Solr 5.0 the more scalable cluster state format now used by default. If a collection was created in any Solr 5.x version or higher, then executing this command is not necessary.

Backup Collection

Backup Solr collections and it's associated configurations to a shared filesystem - for example a Network File System

`/admin/collections?action=BACKUP&name=myBackupName&collection=myCollectionName&location=/path/to/my/shared/drive`

The backup command will backup Solr indexes and configurations for a specified collection. The backup command takes one copy from each shard for the indexes. For configurations it backs up the configSet that was associated with the collection and metadata.

Query Parameters

Key	Type	Required	Description

collection	string	Yes	The name of the collection that needs to be backed up
location	string	No	The location on the shared drive for the backup command to write to. Alternately it can be set as a cluster property
async	string	No	Request ID to track this action which will be processed asynchronously
repository	string	No	The name of the repository to be used for the backup. If no repository is specified then the local filesystem repository will be used automatically.

Restore Collection

Restores Solr indexes and associated configurations.

```
/admin/collections?action=RESTORE&name=myBackupName&location=/path/to/my/sharded/drive&collection=myRestoredCollectionName
```

The restore operation will create a collection with the specified name in the collection parameter. You cannot restore into the same collection the backup was taken from and the target collection should not be present at the time the API is called as Solr will create it for you.

The collection created will be of the same number of shards and replicas as the original collection, preserving routing information, etc. Optionally, you can override some parameters documented below. While restoring, if a configSet with the same name exists in ZooKeeper then Solr will reuse that, or else it will upload the backed up configSet in ZooKeeper and use that.

You can use the collection alias API to make sure client's don't need to change the endpoint to query or index against the newly restored collection.

Query Parameters

Key	Type	Required	Description
collection	string	Yes	The collection where the indexes will be restored into.
location	string	No	The location on the shared drive for the restore command to read from. Alternately it can be set as a cluster property.
async	string	No	Request ID to track this action which will be processed asynchronously.
repository	string	No	The name of the repository to be used for the backup. If no repository is specified then the local filesystem repository will be used automatically.

Additionally, there are several parameters that can be overridden:

Override Parameters

Key	Type	Required	Description
collection.configName	String	No	Defines the name of the configurations to use for this collection. These must already be stored in ZooKeeper. If not provided, Solr will default to the collection name as the configuration name.
replicationFactor	Integer	No	The number of replicas to be created for each shard.

maxShardsPerNode	Integer	No	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.
autoAddReplicas	Boolean	No	When set to true, enables auto addition of replicas on shared file systems. See the section Automatically Add Replicas in SolrCloud for more details on settings and overrides.
property.*name*=*value*	String	No	Set core property *name* to *value*. See the section Defining core.properties for details on supported properties and values.

Delete Replicas in a node

`/admin/collections?action=DELETENODE&node=<node-name>`

Deletes all replicas of all collections in that node. Please note that the node itself will remain as a live node after this operation.

Query Parameters

Key	Type	Required	Description
node	string	Yes	The node to be cleaned up
async	string	No	Request ID to track this action which will be processed asynchronously.

Move all replicas in a node to another

`/admin/collections?action=RELACENODE&source=<source-node>&target=<target-node>`

This command recreates replicas in the source node to the target node. After each replica is copied, the replicas in the source node are deleted

Query Parameters

Key	Type	Required	Description
source	string	Yes	The source node from which the replicas need to be copied from
target	string	Yes	The target node
parallel	boolean	No	default=false. if this flag is set to true, all replicas are created inseparatee threads. Keep in mind that this can lead to very high network and disk I/O if the replicas have very large indices.
async	string	No	Request ID to track this action which will be processed asynchronously.



This operation does not hold necessary locks on the replicas that belong to on the source node. So don't perform other collection operations in this period.

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 requested using the REQUESTSTATUS call at any time.

As of now, REQUESTSTATUS does not automatically clean up the tracking data structures, meaning the status of completed or failed tasks stays stored in ZooKeeper unless cleared manually. DELETESTATUS can be used to clear the stored statuses. However, there is a limit of 10,000 on the number of async call responses stored in a cluster.

Example

Input

```
http://localhost:8983/solr/admin/collections?action=SPLITSHARD&collection=collection
1&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 the `host` and `hostContext` parameters 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 port specified via `bin/solr -p <port>`, or 8983 if not specified.	The port that Solr is running on. This value is only used when `-DzkRun` is specified without a value (see below), to calculate the default port on which embedded ZooKeeper will run. In the `solr.xml` shipped with Solr, the `hostPort` system property is not referenced, and so is ignored. If you want to run Solr on a non-default port, use `bin/solr -p <port>` rather than specifying `-DhostPort`.

`hostContext`	Defaults to `solr`	The context path for the Solr web application.

SolrCloud Instance ZooKeeper Parameters

`zkRun`	Defaults to `localhost:<hostPort+1000>`	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` (with no value) 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

`shard`	Defaults to being automatically assigned based on numShards	Specifies which shard this core acts as a replica of.

`shard` can be specified in the `core.properties` for each core.

Additional cloud related parameters are discussed in Format of 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 at http://localhost:8983/solr/#/~cloud), as well as a few tools available via a ZooKeeper Command Line Utility (CLI). The CLI scripts found in `server/scripts/cloud-scripts` let 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.



Solr's zkcli.sh vs ZooKeeper's zkCli.sh
The `zkcli.sh` provided by Solr is not the same as the `zkCli.sh` included in ZooKeeper distributions.

ZooKeeper's `zkCli.sh` provides a completely general, application-agnostic shell for manipulating data in ZooKeeper. Solr's `zkcli.sh` – discussed in this section – is specific to Solr, and has command line arguments specific to dealing with Solr data in ZooKeeper.

Using Solr's ZooKeeper CLI

Both `zkcli.sh` (for Unix environments) and `zkcli.bat` (for Windows environments) support the following command line options:

Short	Parameter Usage	Meaning
	`-cmd <arg>`	CLI Command to be executed: `bootstrap`, `upconfig`, `downconfig`, `linkconfig`, `makepath`, `get`, `getfile`, `put`, `putfile`, `list`, `clear` or `clusterprop`. This parameter is **mandatory**
`-z`	`-zkhost <locations>`	ZooKeeper host address. 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. For downconfig: the destination of files pulled from Zookeeper
`-h`	`-help`	Display help text.
`-n`	`-confname <arg>`	For `upconfig`, `linkconfig`, `downconfig`: 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.
	`-name <name>`	For `clusterprop`: the **mandatory** cluster property name.
	`-val <value>`	For `clusterprop`: the cluster property value. If not specified, **null** will be used as value.

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.sh` CLI which assume you have already started the SolrCloud example (`bin/solr -e cloud -noprompt`)

If you are on Windows machine, simply replace `zkcli.sh` with `zkcli.bat` in these examples.

Upload a configuration directory

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
   -cmd upconfig -confname my_new_config -confdir
server/solr/configsets/basic_configs/conf
```

Bootstrap ZooKeeper from existing SOLR_HOME

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
   -cmd bootstrap -solrhome /var/solr/data
```

Bootstrap with chroot
Using the boostrap command with a zookeeper chroot in the -zkhost parameter, e.g. `-zkhost 127.0.0.1:2181/solr`, will automatically create the chroot path before uploading the configs.

Put arbitrary data into a new ZooKeeper file

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
   -cmd put /my_zk_file.txt 'some data'
```

Put a local file into a new ZooKeeper file

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
   -cmd putfile /my_zk_file.txt /tmp/my_local_file.txt
```

Link a collection to a configuration set

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:9983 \
   -cmd linkconfig -collection gettingstarted -confname my_new_config
```

Create a new ZooKeeper path

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
   -cmd makepath /solr
```

This can be useful to create a chroot path in ZooKeeper before first cluster start.

Set a cluster property

This command will add or modify a single cluster property in `/clusterprops.json`. Use this command instead of the usual getfile -> edit -> putfile cycle. Unlike the CLUSTERPROP REST API, this command does **not** require a running Solr cluster.

```
./server/scripts/cloud-scripts/zkcli.sh -zkhost 127.0.0.1:2181 \
   -cmd clusterprop -name urlScheme -val https
```

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. 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.

ConfigSets API

The ConfigSets API enables you to create, delete, and otherwise manage ConfigSets. To use a ConfigSet created with this API as the configuration for a collection, use the Collections API.

This API can only be used with Solr running in SolrCloud mode. If you are not running Solr in SolrCloud mode but would still like to use shared configurations, please see the section Config Sets.

API Entry Points

The base URL for all API calls is `http://<hostname>:<port>/solr`.

`/admin/configs?action=CREATE`: create a ConfigSet, based on an existing ConfigSet

`/admin/configs?action=DELETE`: delete a ConfigSet
`/admin/configs?action=LIST`: list all ConfigSets

Create a ConfigSet

`/admin/configs?action=CREATE&name=*name*&baseConfigSet=*baseConfigSet*`

Create a ConfigSet, based on an existing ConfigSet.

Input

Key	Type	Required	Default	Description
name	String	Yes		ConfigSet to be created
baseConfigSet	String	Yes		ConfigSet to copy as a base
configSetProp.*name=value*	String	No		ConfigSet property from base to override

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 a ConfigSet named 'myConfigSet' based on a 'predefinedTemplate' ConfigSet, overriding the immutable property to false.

```
http://localhost:8983/solr/admin/configs?action=CREATE&name=myConfigSet&baseConfigSet=predefinedTemplate&configSetProp.immutable=false
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">323</int>
  </lst>
</response>
```

Delete a ConfigSet

`/admin/configs?action=DELETE&name=*name*`

Delete a ConfigSet

Input

Query Parameters

Key	Type	Required	Default	Description

name	String	Yes		ConfigSet 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 ConfigSet 'myConfigSet'

```
http://localhost:8983/solr/admin/configs?action=DELETE&name=myConfigSet
```

Output

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">170</int>
  </lst>
</response>
```

List ConfigSets

`/admin/configs?action=LIST`

Fetch the names of the ConfigSets in the cluster.

Examples

Input

```
http://localhost:8983/solr/admin/configs?action=LIST&wt=json
```

Output

```
{
  "responseHeader":{
    "status":0,
    "QTime":203},
  "configSets":["myConfigSet1",
    "myConfig2"]}
```

Rule-based Replica Placement

When Solr needs to assign nodes to collections, it can either automatically assign them randomly or the user can specify a set of nodes where it should create the replicas. With very large clusters, it is hard to specify exact

node names and it still does not give you fine grained control over how nodes are chosen for a shard. The user should be in complete control of where the nodes are allocated for each collection, shard and replica. This helps to optimally allocate hardware resources across the cluster.

Rule-based replica assignment allows the creation of rules to determine the placement of replicas in the cluster. In the future, this feature will help to automatically add or remove replicas when systems go down, or when higher throughput is required. This enables a more hands-off approach to administration of the cluster.

This feature is used in the following instances:

- Collection creation
- Shard creation
- Replica creation
- Shard splitting

Common Use Cases

There are several situations where this functionality may be used. A few of the rules that could be implemented are listed below:

- Don't assign more than 1 replica of this collection to a host.
- Assign all replicas to nodes with more than 100GB of free disk space or, assign replicas where there is more disk space.
- Do not assign any replica on a given host because I want to run an overseer there.
- Assign only one replica of a shard in a rack.
- Assign replica in nodes hosting less than 5 cores.
- Assign replicas in nodes hosting the least number of cores.

Rule Conditions

A rule is a set of conditions that a node must satisfy before a replica core can be created there.

Rule Conditions

There are three possible conditions.

- **shard**: this is the name of a shard or a wild card (* means for all shards). If shard is not specified, then the rule applies to the entire collection.
- **replica**: this can be a number or a wild-card (* means any number zero to infinity).
- **tag**: this is an attribute of a node in the cluster that can be used in a rule, e.g. "freedisk", "cores", "rack", "dc", etc. The tag name can be a custom string. If creating a custom tag, a snitch is responsible for providing tags and values. The section Snitches below describes how to add a custom tag, and defines six pre-defined tags (cores, freedisk, host, port, node, and sysprop).

Rule Operators

A condition can have one of the following operators to set the parameters for the rule.

- **equals (no operator required)**: tag:x means tag value must be equal to 'x'
- **greater than (>)**: tag:>x means tag value greater than 'x'. x must be a number
- **less than (<)**: tag:<x means tag value less than 'x'. x must be a number
- **not equal (!)**: tag:!x means tag value MUST NOT be equal to 'x'. The equals check is performed on String value

Fuzzy Operator (~)

This can be used as a suffix to any condition. This would first try to satisfy the rule strictly. If Solr can't find enough nodes to match the criterion, it tries to find the next best match which may not satisfy the criterion. For example, if we have a rule such as, `freedisk:>200~`, Solr will try to assign replicas of this collection on nodes with more than 200GB of free disk space. If that is not possible, the node which has the most free disk space will be chosen instead.

Choosing Among Equals

The nodes are sorted first and the rules are used to sort them. This ensures that even if many nodes match the rules, the best nodes are picked up for node assignment. For example, if there is a rule such as `freedisk:>20`, nodes are sorted first on disk space descending and the node with the most disk space is picked up first. Or, if the rule is `cores:<5`, nodes are sorted with number of cores ascending and the node with the least number of cores is picked up first.

Rules for new shards

The rules are persisted along with collection state. So, when a new replica is created, the system will assign replicas satisfying the rules. When a new shard is created as a result of create shard ensure that you have created rules specific for that shard name. Rules can be altered using the modify collection command. However, it is not required to do so if the rules do not specify explicit shard names. For example, a rule such as `shard:shard1,replica:*,ip_3:168:`, will not apply to any new shard created. But, if your rule is `replica:*,ip_3:168`, then it will apply to any new shard created.

The same is applicable to shard splitting. Shard splitting is treated exactly the same way as shard creation. Even though `shard1_1` and `shard1_2` may be created from `shard1`, the rules treat them as distinct, unrelated shards.

Snitches

Tag values come from a plugin called Snitch. If there is a tag named 'rack' in a rule, there must be Snitch which provides the value for 'rack' for each node in the cluster. A snitch implements the Snitch interface. Solr, by default, provides a default snitch which provides the following tags:

- **cores**: Number of cores in the node
- **freedisk**: Disk space available in the node
- **host**: host name of the node
- **port**: port of the node
- **node**: node name
- **role** : The role of the node. The only supported role is 'overseer'
- **ip_1, ip_2, ip_3, ip_4**: These are ip fragments for each node. For example, in a host with ip `192.168.1.2`, `ip_1 = 2`, `ip_2 =1`, `ip_3 = 168` and `ip_4 = 192`
- **sysprop.{PROPERTY_NAME}**: These are values available from system properties. `sysprop.key` means a value that is passed to the node as `-Dkey=keyValue` during the node startup. It is possible to use rules like `sysprop.key:expectedVal,shard:*`

How Snitches are Configured

It is possible to use one or more snitches for a set of rules. If the rules only need tags from default snitch it need not be explicitly configured. For example:

```
snitch=class:fqn.ClassName,key1:val1,key2:val2,key3:val3
```

How Tag Values are Collected

1. Identify the set of tags in the rules
2. Create instances of Snitches specified. The default snitch is always created.
3. Ask each Snitch if it can provide values for the any of the tags. If even one tag does not have a snitch, the assignment fails.
4. After identifying the Snitches, they provide the tag values for each node in the cluster.
5. If the value for a tag is not obtained for a given node, it cannot participate in the assignment.

Examples

Keep less than 2 replicas (at most 1 replica) of this collection on any node

For this rule, we define the `replica` condition with operators for "less than 2", and use a pre-defined tag named `node` to define nodes with any name.

```
replica:<2,node:*
// this is equivalent to replica:<2,node:*,shard:**. We can omit shard:** because **
is the default value of shard
```

For a given shard, keep less than 2 replicas on any node

For this rule, we use the `shard` condition to define any shard , the `replica` condition with operators for "less than 2", and finally a pre-defined tag named `node` to define nodes with any name.

```
shard:*,replica:<2,node:*
```

Assign all replicas in shard1 to rack 730

This rule limits the `shard` condition to 'shard1', but any number of replicas. We're also referencing a custom tag named `rack`. Before defining this rule, we will need to configure a custom Snitch which provides values for the tag `rack`.

```
shard:shard1,replica:*,rack:730
```

In this case, the default value of `replica` is * (or, all replicas). So, it can be omitted and the rule can be reduced to:

```
shard:shard1,rack:730
```

Create replicas in nodes with less than 5 cores only

This rule uses the `replica` condition to define any number of replicas, but adds a pre-defined tag named `core` and uses operators for "less than 5".

```
replica:*,cores:<5
```

Again, we can simplify this to use the default value for `replica`, like so:

```
cores:<5
```

Do not create any replicas in host 192.45.67.3

This rule uses only the pre-defined tag `host` to define an IP address where replicas should not be placed.

```
host:!192.45.67.3
```

Defining Rules

Rules are specified per collection during collection creation as request parameters. It is possible to specify multiple ‘rule’ and ‘snitch’ params as in this example:

```
snitch=class:EC2Snitch&rule=shard:*,replica:1,dc:dc1&rule=shard:*,replica:<2,dc:dc3
```

These rules are persisted in `clusterstate.json` in Zookeeper and are available throughout the lifetime of the collection. This enables the system to perform any future node allocation without direct user interaction. The rules added during collection creation can be modified later using the MODIFYCOLLECTION API.

Cross Data Center Replication (CDCR)

The SolrCloud architecture is not particularly well suited for situations where a single SolrCloud cluster consists of nodes in separated data clusters connected by an expensive pipe. The root problem is that SolrCloud is designed to support Near Real Time Searching by immediately forwarding updates between nodes in the cluster on a per-shard basis. "CDCR" features exist to help mitigate the risk of an entire Data Center outage.

- What is CDCR?
- Glossary
- Architecture
- Major Components
  - CDCR Configuration
  - CDCR Initialization
  - Inter-Data Center Communication
  - Updates Tracking & Pushing
  - Synchronization of Update Checkpoints
  - Maintenance of Updates Log
  - Monitoring
  - CDC Replicator
  - Limitations
- Configuration
  - Source Configuration
  - Target Configuration
  - Configuration Details
    - The Replica Element
    - The Replicator Element
    - The updateLogSynchronizer Element
    - The Buffer Element
- CDCR API

    - API Entry Points (Control)
    - API Entry Points (Monitoring)
    - Control Commands
    - Monitoring commands
- Initial Startup
- Monitoring
- ZooKeeper settings
- Upgrading and Patching Production

What is CDCR?

The goal of the project is to replicate data to multiple Data Centers. The initial version of the solution will cover the active-passive scenario where data updates are replicated from a Source Data Center to one or more Target Data Centers. The Target Data Center(s) will not propagate updates to the Source Data Center and updates should *not* be sent to any of the Target Data Center(s). Data updates include adds, updates and deletes. Source and Target Data Centers can serve search queries when CDCR is operating. The Target Data Centers will have slightly stale views of the corpus due to propagation delays, but this is minimal (perhaps a few seconds).

Data changes on the Source Data Center are replicated to the Target Data Center only after they are persisted to disk. The data changes can be replicated in real-time (with a small delay) or could be scheduled to be sent in intervals to the Target Data Center. This solution pre-supposes that the Source and Target data centers begin with the same documents indexed. Of course the indexes may be empty to start.

Each shard leader in the Source Data Center will be responsible for replicating its updates to the appropriate collection in the Target Data Center. When receiving updates from the Source Data Center, shard leaders in the Target Data Center will replicate the changes to their own replicas.

This replication model is designed to tolerate some degradation in connectivity, accommodate limited bandwidth, and support batch updates to optimize communication.

Replication supports both a new empty index and pre-built indexes. In the scenario where the replication is set up on a pre-built index, CDCR will ensure consistency of the replication of the updates, but cannot ensure consistency on the full index. Therefore any index created before CDCR was set up will have to be replicated by other means (described in the section Starting CDCR the first time with an existing index) in order that Source and Target indexes be fully consistent.

The active-passive nature of the initial implementation implies a "push" model from the Source collection to the Target collection. Therefore, the Source configuration must be able to "see" the ZooKeeper ensemble in the Target cluster. The ZooKeeper ensemble is provided configured in the Source's `solrconfig.xml` file.

CDCR is configured to replicate from collections in the Source cluster to collections in the Target cluster on a collection-by-collection basis. Since CDCR is configured in `solrconfig.xml` (on both Source and Target clusters), the settings can be tailored for the needs of each collection.

CDCR can be configured to replicate from one collection to a second collection *within the same cluster*. That is a specialized scenario not covered in this document.

Glossary

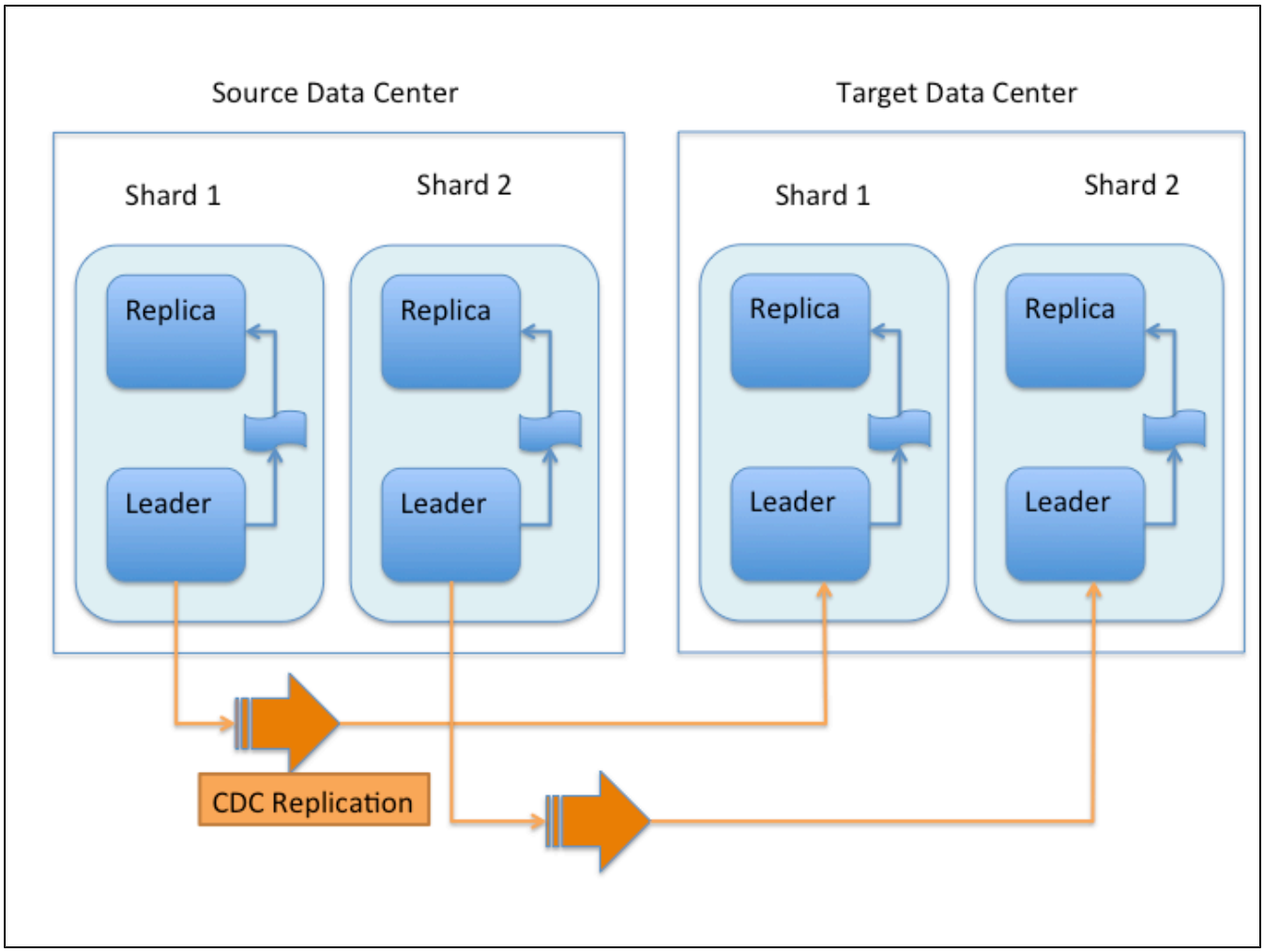
Terms used in this document include:

- **Node**: A JVM instance running Solr; a server.
- **Cluster**: A set of Solr nodes managed as a single unit by a ZooKeeper ensemble, hosting one or more Collections.
- **Data Center:** A group of networked servers hosting a Solr cluster. In this document, the terms *Cluster* and *Data Center* are interchangeable as we assume that each Solr cluster is hosted in a different group of networked servers.

- **Shard**: A sub-index of a single logical collection. This may be spread across multiple nodes of the cluster. Each shard can have as many replicas as needed.
- **Leader**: Each shard has one node identified as its leader. All the writes for documents belonging to a shard are routed through the leader.
- **Replica**: A copy of a shard for use in failover or load balancing. Replicas comprising a shard can either be leaders or non-leaders.
- **Follower:** A convenience term for a replica that is *not* the leader of a shard.
- **Collection**: Multiple documents that make up one logical index. A cluster can have multiple collections.
- **Updates Log**: An append-only log of write operations maintained by each node.

Architecture

Here is a picture of the data flow.



Updates and deletes are first written to the Source cluster, then forwarded to the Target cluster. The data flow sequence is:

1. A shard leader receives a new data update that is processed by its Update Processor.
2. The data update is first applied to the local index.
3. Upon successful application of the data update on the local index, the data update is added to the Updates Log queue.
4. After the data update is persisted to disk, the data update is sent to the replicas within the Data Center.
5. After Step 4 is successful CDCR reads the data update from the Updates Log and pushes it to the corresponding collection in the Target Data Center. This is necessary in order to ensure consistency between the Source and Target Data Centers.

6. The leader on the Target data center writes the data locally and forwards it to all its followers.

Steps 1, 2, 3 and 4 are performed synchronously by SolrCloud; Step 5 is performed asynchronously by a background thread. Given that CDCR replication is performed asynchronously, it becomes possible to push batch updates in order to minimize network communication overhead. Also, if CDCR is unable to push the update at a given time -- for example, due to a degradation in connectivity -- it can retry later without any impact on the Source Data Center.

One implication of the architecture is that the leaders in the Source cluster must be able to "see" the leaders in the Target cluster. Since leaders may change, this effectively means that all nodes in the Source cluster must be able to "see" all Solr nodes in the Target cluster so firewalls, ACL rules, etc. must be configured with care.

Major Components

There are a number of key features and components in CDCR's architecture:

CDCR Configuration

In order to configure CDCR, the Source Data Center requires the host address of the ZooKeeper cluster associated with the Target Data Center. The ZooKeeper host address is the only information needed by CDCR to instantiate the communication with the Target Solr cluster. The CDCR configuration file on the Source cluster will therefore contain a list of ZooKeeper hosts. The CDCR configuration file might also contain secondary/optional configuration, such as the number of CDC Replicator threads, batch updates related settings, etc.

CDCR Initialization

CDCR supports incremental upddates to either new or existing collections. CDCR may not be able to keep up with very high volume updates, especially if there are significant communications latencies due to a slow "pipe" between the data centers. Some scenarios:

- There is an initial bulk load of a corpus followed by lower volume incremental updates. In this case, one can do the initial bulk load and then enable CDCR. See the section Starting CDCR the first time with an existing index for more information.
- The index is being built up from scratch, without a significant initial bulk load. CDCR can be set up on empty collections and keep them synchronized from the start.
- The index is always being updated at a volume too high for CDCR to keep up. This is especially possible in situations where the connection between the Source and Target data centers is poor. This scenario is unsuitable for CDCR in its current form.

Inter-Data Center Communication

Communication between Data Centers will be achieved through HTTP and the Solr REST API using the SolrJ client. The SolrJ client will be instantiated with the ZooKeeper host of the Target Data Center. SolrJ will manage the shard leader discovery process.

Updates Tracking & Pushing

CDCR replicates data updates from the Source to the Target Data Center by leveraging the Updates Log.

A background thread regularly checks the Updates Log for new entries, and then forwards them to the Target Data Center. The thread therefore needs to keep a checkpoint in the form of a pointer to the last update successfully processed in the Updates Log. Upon acknowledgement from the Target Data Center that updates have been successfully processed, the Updates Log pointer is updated to reflect the current checkpoint.

This pointer must be synchronized across all the replicas. In the case where the leader goes down and a new leader is elected, the new leader will be able to resume replication from the last update by using this synchronized pointer. The strategy to synchronize such a pointer across replicas will be explained next.

If for some reason, the Target Data Center is offline or fails to process the updates, the thread will periodically try to contact the Target Data Center and push the updates.

Synchronization of Update Checkpoints

A reliable synchronization of the update checkpoints between the shard leader and shard replicas is critical to avoid introducing inconsistency between the Source and Target Data Centers. Another important requirement is that the synchronization must be performed with minimal network traffic to maximize scalability.

In order to achieve this, the strategy is to:

- Uniquely identify each update operation. This unique identifier will serve as pointer.
- Rely on two storages: an ephemeral storage on the Source shard leader, and a persistent storage on the Target cluster.

The shard leader in the Source cluster will be in charge of generating a unique identifier for each update operation, and will keep a copy of the identifier of the last processed updates in memory. The identifier will be sent to the Target cluster as part of the update request. On the Target Data Center side, the shard leader will receive the update request, store it along with the unique identifier in the Updates Log, and replicate it to the other shards.

SolrCloud is already providing a unique identifier for each update operation, i.e., a “version” number. This version number is generated using a time-based lmport clock which is incremented for each update operation sent. This provides an “happened-before” ordering of the update operations that will be leveraged in (1) the initialization of the update checkpoint on the Source cluster, and in (2) the maintenance strategy of the Updates Log.

The persistent storage on the Target cluster is used only during the election of a new shard leader on the Source cluster. If a shard leader goes down on the Source cluster and a new leader is elected, the new leader will contact the Target cluster to retrieve the last update checkpoint and instantiate its ephemeral pointer. On such a request, the Target cluster will retrieve the latest identifier received across all the shards, and send it back to the Source cluster. To retrieve the latest identifier, every shard leader will look up the identifier of the first entry in its Update Logs and send it back to a coordinator. The coordinator will have to select the highest among them.

This strategy does not require any additional network traffic and ensures reliable pointer synchronization. Consistency is principally achieved by leveraging SolrCloud. The update workflow of SolrCloud ensures that every update is applied to the leader but also to any of the replicas. If the leader goes down, a new leader is elected. During the leader election, a synchronization is performed between the new leader and the other replicas. As a result, this ensures that the new leader has a consistent Update Logs with the previous leader. Having a consistent Updates Log means that:

- On the Source cluster, the update checkpoint can be reused by the new leader.
- On the Target cluster, the update checkpoint will be consistent between the previous and new leader. This ensures the correctness of the update checkpoint sent by a newly elected leader from the Target cluster.

Maintenance of Updates Log

The CDCR replication logic requires modification to the maintenance logic of the Updates Log on the Source Data Center. Initially, the Updates Log acts as a fixed size queue, limited to 100 update entries. In the CDCR scenario, the Update Logs must act as a queue of variable size as they need to keep track of all the updates up through the last processed update by the Target Data Center. Entries in the Update Logs are removed only when all pointers (one pointer per Target Data Center) are after them.

If the communication with one of the Target Data Center is slow, the Updates Log on the Source Data Center can grow to a substantial size. In such a scenario, it is necessary for the Updates Log to be able to efficiently find a given update operation given its identifier. Given that its identifier is an incremental number, it is possible to

implement an efficient search strategy. Each transaction log file contains as part of its filename the version number of the first element. This is used to quickly traverse all the transaction log files and find the transaction log file containing one specific version number.

Monitoring

CDCR provides the following monitoring capabilities over the replication operations:

- Monitoring of the outgoing and incoming replications, with information such as the Source and Target nodes, their status, etc.
- Statistics about the replication, with information such as operations (add/delete) per second, number of documents in the queue, etc.

Information about the lifecycle and statistics will be provided on a per-shard basis by the CDC Replicator thread. The CDCR API can then aggregate this information an a collection level.

CDC Replicator

The CDC Replicator is a background thread that is responsible for replicating updates from a Source Data Center to one or more Target Data Centers. It will also be responsible in providing monitoring information on a per-shard basis. As there can be a large number of collections and shards in a cluster, we will use a fixed-size pool of CDC Replicator threads that will be shared across shards.

Limitations

The current design of CDCR has some limitations. CDCR will continue to evolve over time and many of these limitations will be addressed. Among them are:

- CDCR is unlikely to be satisfactory for bulk-load situations where the update rate is high, especially if the bandwidth between the Source and Target clusters is restricted. In this scenario, the initial bulk load should be performed, the Source and Target data centers synchronized and CDCR be utilized for incremental updates.
- CDCR is currently only active-passive; data is pushed from the Source cluster to the Target cluster. There is active work being done in this area in the 6x code line to remove this limitation.

Configuration

The Source and Target configurations differ in the case of the data centers being in separate clusters. "Cluster" here means separate ZooKeeper ensembles controlling disjoint Solr instances. Whether these data centers are physically separated or not is immaterial for this discussion.

Source Configuration

Here is a sample of a Source configuration file, a section in `solrconfig.xml`. The presence of the <replica> section causes CDCR to use this cluster as the Source and should not be present in the Target collections in the cluster-to-cluster case. Details about each setting are after the two examples:

```
<requestHandler name="/cdcr" class="solr.CdcrRequestHandler">
  <lst name="replica">
    <str name="zkHost">10.240.18.211:2181</str>
    <str name="source">collection1</str>
    <str name="target">collection1</str>
  </lst>

  <lst name="replicator">
    <str name="threadPoolSize">8</str>
    <str name="schedule">1000</str>
    <str name="batchSize">128</str>
  </lst>

  <lst name="updateLogSynchronizer">
    <str name="schedule">1000</str>
  </lst>
</requestHandler>

<!-- Modify the <updateLog> section of your existing <updateHandler>
     in your config as below -->
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog class="solr.CdcrUpdateLog">
    <str name="dir">${solr.ulog.dir:}</str>
    <!--Any parameters from the original <updateLog> section -->
  </updateLog>
</updateHandler>
```

Target Configuration

Here is a typical Target configuration.

Target instance must configure an update processor chain that is specific to CDCR. The update processor chain must include the **CdcrUpdateProcessorFactory**. The task of this processor is to ensure that the version numbers attached to update requests coming from a CDCR Source SolrCloud are reused and not overwritten by the Target. A properly configured Target configuration looks similar to this.

```
<requestHandler name="/cdcr" class="solr.CdcrRequestHandler">
  <lst name="buffer">
    <str name="defaultState">disabled</str>
  </lst>
</requestHandler>

<requestHandler name="/update" class="solr.UpdateRequestHandler">
  <lst name="defaults">
    <str name="update.chain">cdcr-processor-chain</str>
  </lst>
</requestHandler>

<updateRequestProcessorChain name="cdcr-processor-chain">
  <processor class="solr.CdcrUpdateProcessorFactory"/>
  <processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>

<!-- Modify the <updateLog> section of your existing <updateHandler> in your
    config as below -->
<updateHandler class="solr.DirectUpdateHandler2">
  <updateLog class="solr.CdcrUpdateLog">
    <str name="dir">${solr.ulog.dir:}</str>
    <!--Any parameters from the original <updateLog> section -->
  </updateLog>
</updateHandler>
```

Configuration Details

The configuration details, defaults and options are as follows:

The Replica Element

CDCR can be configured to forward update requests to one or more replicas. A replica is defined with a “replica” list as follows:

Parameter	Required	Default	Description
zkHost	Yes	none	The host address for ZooKeeper of the Target SolrCloud. Usually this is a comma-separated list of addresses to each node in the Target ZooKeeper ensemble.
Source	Yes	none	The name of the collection on the Source SolrCloud to be replicated.
Target	Yes	none	The name of the collection on the Target SolrCloud to which updates will be forwarded.

The Replicator Element

The CDC Replicator is the component in charge of forwarding updates to the replicas. The replicator will monitor the update logs of the Source collection and will forward any new updates to the Target collection. The replicator uses a fixed thread pool to forward updates to multiple replicas in parallel. If more than one replica is configured, one thread will forward a batch of updates from one replica at a time in a round-robin fashion. The replicator can be configured with a “replicator” list as follows:

Parameter	Required	Default	Description

threadPoolSize	No	2	The number of threads to use for forwarding updates. One thread per replica is recommended.
schedule	No	10	The delay in milliseconds for the monitoring the update log(s).
batchSize	No	128	The number of updates to send in one batch. The optimal size depends on the size of the documents. Large batches of large documents can increase your memory usage significantly.

The updateLogSynchronizer Element

Expert: Non-leader nodes need to synchronize their update logs with their leader node from time to time in order to clean deprecated transaction log files. By default, such a synchronization process is performed every minute. The schedule of the synchronization can be modified with a “updateLogSynchronizer” list as follows:

Parameter	Required	Default	Description
schedule	No	60000	The delay in milliseconds for synchronizing the updates log.

The Buffer Element

CDCR is configured by default to buffer any new incoming updates. When buffering updates, the updates log will store all the updates indefinitely. Replicas do not need to buffer updates, and it is recommended to disable buffer on the Target SolrCloud. The buffer can be disabled at startup with a “buffer” list and the parameter “defaultState” as follows:

Parameter	Required	Default	Description
defaultState	No	enabled	The state of the buffer at startup.

CDCR API

The CDCR API is used to control and monitor the replication process. Control actions are performed at a collection level, i.e., by using the following base URL for API calls: http://<hostname>:<port>/solr/<collection>. Monitor actions are performed at a core level, i.e., by using the following base URL for API calls: http://<hostname>:<port>/solr/<collection>.

Currently, none of the CDCR API calls have parameters.

API Entry Points (Control)

`collection/cdcr?action=STATUS`: Returns the current state of CDCR.
`collection/cdcr?action=START`: Starts CDCR replication
`collection/cdcr?action=STOPPED`: Stops CDCR replication.
`collection/cdcr?action=ENABLEBUFFER`: Enables the buffering of updates.
`collection/cdcr?action=DISABLEBUFFER`: Disables the buffering of updates.

API Entry Points (Monitoring)

`core/cdcr?action=QUEUES`: Fetches statistics about the queue for each replica and about the update logs.
`core/cdcr?action=OPS`: Fetches statistics about the replication performance (operations per second) for each replica
`core/cdcr?action=ERRORS`: Fetches statistics and other information about replication errors for each replica.

Control Commands

`/collection/cdcr?action=STATUS`

Input

Query Parameters: There are no parameters to this command.

Output

Output Content

The current state of the CDCR, which includes the state of the replication process and the state of the buffer.

Examples

Input: There are no parameters to this command.

```
http://host:8983/solr/<collection_name>/cdcr?action=STATUS
```

Output

```
{
  "responseHeader": {
  "status": 0,
  "QTime": 0
  },
  "status": {
  "process": "stopped",
  "buffer": "enabled"
  }
}
```

`/collection/cdcr?action=ENABLEBUFFER`

Input

Query Parameters: There are no parameters to this command.

Output

Output Content

The status of the process and an indication of whether the buffer is enabled

Examples

Input This command enables the buffer, there are no parameters.

```
http://host:8983/solr/<collection_name>/cdcr?action=ENABLEBUFFER
```

Output

```
{
  "responseHeader": {
  "status": 0,
  "QTime": 0
  },
  "status": {
  "process": "started",
  "buffer": "enabled"
  }
}
```

`/collection/cdcr?action=DISABLEBUFFER`

Input

Query Parameters: There are no parameters to this command

Output

Output Content: The status of CDCR and an indication that the buffer is disabled.

Examples

Input: This command disables buffering

```
http://host:8983/solr/<collection_name>/cdcr?action=DISABLEBUFFER
```

Output: The status of CDCR and an indication that the buffer is disabled.

```
{
  "responseHeader": {
  "status": 0,
  "QTime": 0
  },
  "status": {
  "process": "started",
  "buffer": "disabled"
  }
}
```

`/collection/cdcr?action=START`

Input

Query Parameters: There are no parameters for this action

Output

Output Content: Confirmation that CDCR is started and the status of buffering

Examples

Input

```
http://host:8983/solr/<collection_name>/cdcr?action=START
```

Output

```
{
  "responseHeader": {
  "status": 0,
  "QTime": 0
  },
  "status": {
  "process": "started",
  "buffer": "enabled"
  }
}
```

`/collection/cdcr?action=STOPPED`

Input

Query Parameters: There are no parameters for this command.

Output

Output Content: The status of CDCR, including the confirmation that CDCR is stopped

Examples

Input

```
 http://host:8983/solr/<collection_name>/cdcr?action=STOPPED
```

Output

```
{
  "responseHeader": {
  "status": 0,
  "QTime": 0
  },
  "status": {
  "process": "stopped",
  "buffer": "enabled"
  }
}
```

Monitoring commands

`/core/cdcr?action=QUEUES`

Input

Query Parameters: There are no parameters for this command

Output

Output Content

The output is composed of a list “queues” which contains a list of (ZooKeeper) Target hosts, themselves containing a list of Target collections. For each collection, the current size of the queue and the timestamp of the

last update operation successfully processed is provided. The timestamp of the update operation is the original timestamp, i.e., the time this operation was processed on the Source SolrCloud. This allows an estimate the latency of the replication process.

The “queues” object also contains information about the updates log, such as the size (in bytes) of the updates log on disk (“tlogTotalSize”), the number of transaction log files (“tlogTotalCount”) and the status of the updates log synchronizer (“updateLogSynchronizer”).

Examples

Input

```
http://host:8983/solr/<replica_name>/cdcr?action=QUEUES
```

Output

```
{
  responseHeader={
    status=0,
    QTime=1
  },
  queues={
    127.0.0.1: 40342/solr={
    Target_collection={
        queueSize=104,
        lastTimestamp=2014-12-02T10: 32: 15.879Z
      }
    }
  },
  tlogTotalSize=3817,
  tlogTotalCount=1,
  updateLogSynchronizer=stopped
}
```

/core/cdcr?action=OPS

Input

Query Parameters: There are no parameters for this command.

Output

Output Content: The output is composed of a list “operationsPerSecond” which contains a list of (ZooKeeper) Target hosts, themselves containing a list of Target collections. For each collection, the average number of processed operations per second since the start of the replication process is provided. The operations are further broken down into two groups: add and delete operations.

Examples

Input

```
http://host:8983/solr/<collection_name>/cdcr?action=OPS
```

Output

```
{
  responseHeader={
    status=0,
    QTime=1
  },
  operationsPerSecond={
    127.0.0.1: 59661/solr={
      Target_collection={
          all=297.102944952749052,
          adds=297.102944952749052,
          deletes=0.0
      }
    }
  }
}
```

`/core/cdcr?action=ERRORS`

Input

Query Parameters: There are no parameters for this command.

Output

Output Content: The output is composed of a list “errors” which contains a list of (ZooKeeper) Target hosts, themselves containing a list of Target collections. For each collection, information about errors encountered during the replication is provided, such as the number of consecutive errors encountered by the replicator thread, the number of bad requests or internal errors since the start of the replication process, and a list of the last errors encountered ordered by timestamp.

Examples

Input

```
http://host:8983/solr/<collection_name>/cdcr?action=ERRORS
```

Output

```
{
  responseHeader={
    status=0,
    QTime=2
  },
  errors={
    127.0.0.1: 36872/solr={
      Target_collection={
        consecutiveErrors=3,
        bad_request=0,
        internal=3,
        last={
          2014-12-02T11: 04: 42.523Z=internal,
          2014-12-02T11: 04: 39.223Z=internal,
          2014-12-02T11: 04: 38.22Z=internal
        }
      }
    }
  }
}
```

Initial Startup

This is a general approach for initializing CDCR in a production environment based upon an approach taken by the initial working installation of CDCR and generously contributed to illustrate a "real world" scenario. NOTE: The configuration snippets below illustrate specific points of configuration.

- Customer uses the CDCR approach to keep a remote DR instance available for production backup. This is an active-passive solution.
- Customer has 26 clouds with 200 million assets per cloud (15GB indexes). Total document count is over 4.8 billion.
    - Source and Target clouds were synched in 2-3 hour maintenance windows to establish the base index for the Targets.
- Tip: As usual, it is good to start small. Sync a single cloud and monitor for a period of time before doing the others. You may need to adjust your settings several times before finding the right balance.
    - Before starting, stop or pause the indexers. This is best done during a small maintenance window.
    - Stop the SolrCloud instances at the Source
    - Include the cdcr request handler configuration in `solrconfig.xml`

```
 http://localhost:898
<requestHandler name="/cdcr" class="solr.CdcrRequestHandler">
    <lst name="replica">
      <str name="zkHost">${TargetZk}</str>
      <str name="Source">${SourceCollection}</str>
      <str name="Target">${TargetCollection}</str>
    </lst>
    <lst name="replicator">
      <str name="threadPoolSize">8</str>
      <str name="schedule">10</str>
      <str name="batchSize">2000</str>
    </lst>
    <lst name="updateLogSynchronizer">
      <str name="schedule">1000</str>
    </lst>
  </requestHandler>

  <updateRequestProcessorChain name="cdcr-processor-chain">
    <processor class="solr.CdcrUpdateProcessorFactory" />
    <processor class="solr.RunUpdateProcessorFactory" />
  </updateRequestProcessorChain>
```

- Upload the modified `solrconfig.xml` to ZooKeeper on both Source and Target
- Sync the index directories from the Source collection to Target collection across to the corresponding shard nodes.
    - Tip: rsync works well for this.

For example: if there are 2 shards on collection1 with 2 replicas for each shard, copy the corresponding index directories from

shard1replica1Source	to	shard1replica1Target
shard1replica2Source	to	shard1replica2Target
shard2replica1Source	to	shard2replica1Target
shard2replica2Source	to	shard2replica2Target

- Start the ZooKeeper on the Target (DR) side
- Start the SolrCloud on the Target (DR) side
- Start the ZooKeeper on the Source side
- Start the SolrCloud on the Source side
    - Tip: As a general rule, the Target (DR) side of the SolrCloud should be started before the Source side.
- Activate the CDCR on Source instance using the cdcr api

http://host:port/solr/collection_name/cdcr?action=START

```
http://host:port/solr/<collection_name>/cdcr?action=START
```

- There is no need to run the /cdcr?action=START command on the Target
- Disable the buffer on the Target

```
http://host:port/solr/collection_name/cdcr?action=DISABLEBUFFER
```

- Renable indexing

Monitoring

1. Network and disk space monitoring are essential. Ensure that the system has plenty of available storage to queue up changes if there is a disconnect between the Source and Target. A network outage between the two data centers can cause your disk usage to grow.
    a. Tip: Set a monitor for your disks to send alerts when the disk gets over a certain percentage (eg. 70%)
    b. Tip: Run a test. With moderate indexing, how long can the system queue changes before you run out of disk space?
2. Create a simple way to check the counts between the Source and the Target.
    a. Keep in mind that if indexing is running, the Source and Target may not match document for document. Set an alert to fire if the difference is greater than some percentage of the overall cloud size.

ZooKeeper settings

1. With CDCR, the Target ZooKeepers will have connections from the Target clouds and the Source clouds. You may need to increase the maxClientCnxns setting in the zoo.cfg.

```
## set numbers of connection to 200 from client
## is maxClientCnxns=0 that means no limit
maxClientCnxns=800
```

Upgrading and Patching Production

1. When rolling in upgrades to your indexer or application, you should shutdown the Source (production) and the Target (DR). Depending on your setup, you may want to pause/stop indexing. Deploy the release or patch and renable indexing. Then start the Target (DR).
    a. Tip: There is no need to reissue the DISABLEBUFFERS or START commands. These are persisted.
    b. Tip: After starting the Target, run a simple test. Add a test document to each of the Source clouds. Then check for it on the Target.

```
#send to the Source
curl http://<Source>/solr/cloud1/update -H 'Content-type:application/json' -d
'[{"SKU":"ABC"}]'


#check the Target
curl
"http://<Target>:8983/solr/<collection_name>/select?q=SKU:ABC&wt=json&indent=true"
```

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 Making and Restoring Backups).

Distributed Search with Index Sharding

It is highly recommended that you use SolrCloud when needing to scale up or scale out. The setup described below is legacy and was used prior to the existence of SolrCloud. SolrCloud provides for a truly distributed set of features with support for things like automatic routing, leader election, optimistic concurrency and other sanity checks that are expected out of a distributed system.

Everything on this page is specific to legacy setup of distributed search. Users trying out SolrCloud should not follow any of the steps or information below.

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

When not using SolrCloud, it is up to you to get all your documents indexed on each shard of your server farm. Solr supports distributed indexing (routing) in it's true form only in the SolrCloud mode.

In the legacy distributed mode, Solr does not calculate universal term/doc frequencies. For most large-scale implementations, it is not likely to matter that Solr calculates TF/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.

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/core1/select?shards=solr1:8983/solr/core1,solr2:8983/solr/core1&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.

As in SolrCloud mode, when `shards.info=true`, distributed responses will include information about the shard (where each shard represents a logically different index or physical location)

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.
- 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.
- 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`.

Formerly a limitation was that TF/IDF relevancy computations only used shard-local statistics. This is still the case by default. If your data isn't randomly distributed, or if you want more exact statistics, then remember to configure the ExactStatsCache.

Avoiding Distributed Deadlock

Like in SolrCloud mode, inter-shard requests could lead to a distributed deadlock. It can be avoided by following the instructions here.

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 two Solr home directories:

```
mkdir example/nodes
mkdir example/nodes/node1
# Copy solr.xml into this solr.home
cp server/solr/solr.xml example/nodes/node1/.
# Repeat the above steps for the second node
mkdir example/nodes/node2
cp server/solr/solr.xml example/nodes/node2/.
```

2. Start the two Solr instances

```
# Start first node on port 8983
bin/solr start -s example/nodes/node1 -p 8983

# Start second node on port 8984
bin/solr start -s example/nodes/node2 -p 8984
```

3. Create a core on both the nodes with the sample_techproducts_configs.

```
bin/solr create_core -c core1 -p 8983 -d sample_techproducts_configs
# Create a core on the Solr node running on port 8984
bin/solr create_core -c core1 -p 8984 -d sample_techproducts_configs
```

4. In the third window, index an example document to each of the server:

```
bin/post -c core1 example/exampledocs/monitor.xml -port 8983

bin/post -c core1 example/exampledocs/monitor2.xml -port 8984
```

5. Search on the node on port 8983:

```
curl http://localhost:8983/solr/core1/select?q=*:*&wt=xml&indent=true
```

   This should bring back one document.

   Search on the node on port 8984:

```
curl http://localhost:8984/solr/core1/select?q=*:*&wt=xml&indent=true
```

   This should also bring back a single document.

   Now do a distributed search across both servers with your browser or `curl.` In the example below, an extra parameter 'fl' is passed to restrict the returned fields to id and name.

```
curl
http://localhost:8983/solr/core1/select?q=*:*&indent=true&shards=localhost:898
3/solr/core1,localhost:8984/solr/core1&fl=id,name
```

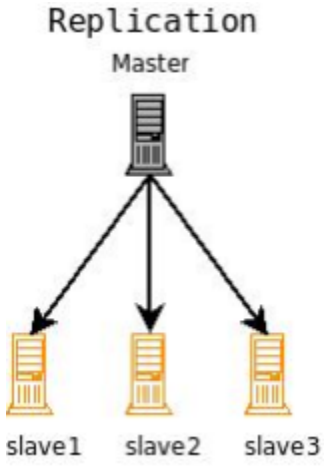
   This should contain both the documents as shown below:

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">8</int>
    <lst name="params">
      <str name="q">*:*</str>
      <str
name="shards">localhost:8983/solr/core1,localhost:8984/solr/core1</str>
      <str name="indent">true</str>
      <str name="fl">id,name</str>
      <str name="wt">xml</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0" maxScore="1.0">
    <doc>
      <str name="id">3007WFP</str>
      <str name="name">Dell Widescreen UltraSharp 3007WFP</str>
    </doc>
    <doc>
      <str name="id">VA902B</str>
      <str name="name">ViewSonic VA902B - flat panel display - TFT - 19"</str>
    </doc>
  </result>
</response>
```

Index Replication

Index Replication distributes complete copies of a master index to one or more slave servers. The master server continues to manage updates to the index. All querying is handled by the slaves. This division of labor enables Solr to scale to provide adequate responsiveness to queries against large search volumes.

The figure below shows a Solr configuration using index replication. The master server's index is replicated on the slaves.



A Solr index can be replicated across multiple slave servers, which then process requests.

Topics covered in this section:

- Index Replication in Solr
- Replication Terminology
- Configuring the ReplicationHandler
- Setting Up a Repeater with the ReplicationHandler
- Commit and Optimize Operations
- Slave Replication
- HTTP API Commands for the ReplicationHandler
- Distribution and Optimization

Index Replication in Solr

Solr includes a Java implementation of index replication that works over HTTP:

- 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 file system features (eg: 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 request handler. Configuring replication is therefore similar to any normal request handler.

Replication In SolrCloud
Although there is no explicit concept of "master/slave" nodes in a SolrCloud cluster, the `ReplicationHandler` discussed on this page is still used by SolrCloud as needed to support "shard recovery" – but this is done in a peer to peer manner. When using SolrCloud, the `ReplicationHandler` must be available via the `/replication` path. Solr does this implicitly unless overridden explicitly in your `solrconfig.xml`, but If you wish to override the default behavior, make certain that you do not explicitly set any of the "master" or "slave" configuration options mentioned below, or they will interfere with normal SolrCloud operation.

Replication Terminology

The table below defines the key terms associated with Solr replication.

Term	Definition
Index	A Lucene index is a directory of files. These files make up the searchable and returnable data of a Solr Core.
Distribution	The copying of an index 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 index, 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 index is optimized.

Master and Slave	A Solr replication master is a single node which receives all updates initially and keeps everything organized. Solr replication slave nodes receive no updates directly, instead all changes (such as inserts, updates, deletes, etc.) are made against the single master node. Changes made on the master are distributed to all the slave nodes 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. Optimization should only be run on the master nodes. An optimized index may give query performance gains compared to an index that has become fragmented over a period of time with many updates. Distributing an optimized index requires a much longer time than the distribution of new segments to an un-optimized index.
Segments	A self contained subset of an index consisting of some documents and data structures related to the inverted index of terms in those documents.
mergeFactor	A parameter that controls the number of segments in an index. 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 of an index. Snapshots are distributed from the master nodes when the slaves pull them, "smart copying" any segments the slave node does not have in snapshot directory that contains the hard links to the most recent index data files.

Configuring the ReplicationHandler

In addition to `ReplicationHandler` configuration options specific to the master/slave roles, there are a few special configuration options that are generally supported (even when using SolrCloud).

- `maxNumberOfBackups` an integer value dictating the maximum number of backups this node will keep on disk as it receives `backup` commands.
- Similar to most other request handlers in Solr you may configure a set of "defaults, invariants, and/or appends" parameters corresponding with any request parameters supported by the `ReplicationHandler` when processing commands.

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 a possible 'master' configuration for the `ReplicationHandler`, including a fixed number of backups and an invariant setting for the `maxWriteMBPerSec` request parameter to prevent slaves from saturating it's network interface

```
<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>
  <lst name="invariants">
    <str name="maxWriteMBPerSec">16</str>
  </lst>
</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/core_name/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>
```

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/core_name/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` dire ctory 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/*core_name*/replication?command=enablereplication	Enables replication on the master for all its slaves.
http://*master_host:port*/solr/*core_name*/replication?command=disablereplication	Disables replication on the master for all its slaves.
http://*host:port*/solr/*core_name*/replication?command=indexversion	Returns the version of the latest replicatable index on the specified master or slave.
http://*slave_host:port*/solr/*core_name*/replication?command=fetchindex	Forces the specified slave to fetch a copy of the index from its master. If you like, you can pass an extra attribute such as masterUrl or compression (or any other parameter which is specified in the `<lst name="slave">` tag) to do a one time replication from a master. This obviates the need for hard-coding the master in the slave.

http://*slave_host:port*/solr/*core_name*/replication?command=abortfetch	Aborts copying an index from a master to the specified slave.
http://*slave_host:port*/solr/*core_name*/replication?command=enablepoll	Enables the specified slave to poll for changes on the master.
http://*slave_host:port*/solr/*core_name*/replication?command=disablepoll	Disables the specified slave from polling for changes on the master.
http://*slave_host:port*/solr/*core_name*/replication?command=details	Retrieves configuration details and current status.
http://*host:port*/solr/*core_name*/replication?command=filelist&generation=<*generation-number*>	Retrieves a list of Lucene files present in the specified host's index. You can discover the generation number of the index by running the `indexversion` command.
http://*master_host:port*/solr/*core_name*/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. supported request parameters: • `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. • `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 • `location` : Backup location
http://*master_host:port* /solr/*core_name*/replication?command=deletebackup	Delete any backup created using the `backup` command . request parameters: • name: The name of the snapshot . A snapshot with the name snapshot.<name> must exist .If not, an error is thrown • location: Location where the snapshot is created

Distribution and Optimization

Optimizing an index is not something most users should generally worry about - but in particular users should be aware of the impacts of optimizing an index when using the `ReplicationHandler`.

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 index 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 index means that the **entire** index 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 index 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 slave node 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* the optimize, 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 indexes 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 index can be distributed in the background as queries are being normally serviced. The optimization can occur at any time convenient to the application providing index updates.

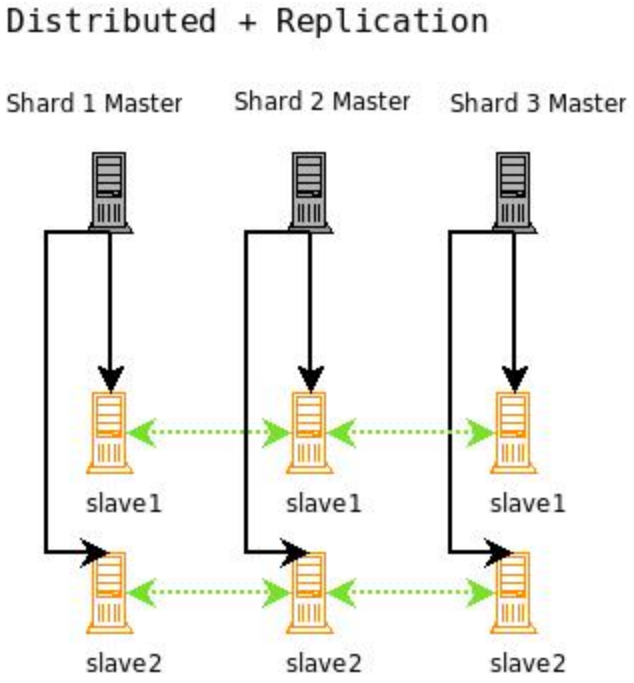
While optimizing may have some benefits in some situations, a rapidly changing index will not retain those benefits for long, and since optimization is an intensive process, it may be better to consider other options, such as lowering the merge factor (discussed in the section on Index Configuration).

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. Make sure that both indexes you want to merge are closed.
2. Issue this command:

```
java -cp $SOLR/server/solr-webapp/webapp/WEB-INF/lib/lucene-core-VERSION.jar:$
SOLR/server/solr-webapp/webapp/WEB-INF/lib/lucene-misc-VERSION.jar
 org/apache/lucene/misc/IndexMergeTool
 /path/to/newindex
 /path/to/old/index1
 /path/to/old/index2
```

   This will create a new index at `/path/to/newindex` that contains both index1 and index2.
3. Copy this new directory to the location of your application's solr index (move the old one aside first, of course) and start Solr.

Using CoreAdmin

The `MERGEINDEXES` command of the CoreAdminHandler can be used to merge indexes into a new core – either from one or more arbitrary `indexDir` directories or by merging from one or more existing `srcCore` core names.

See the CoreAdminHandler section for details.

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 Response 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	https://github.com/rsolr/rsolr
DelSolr	Ruby	https://github.com/avvo/delsolr
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	https://github.com/mausch/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/collection_name/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/collection_name/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 `SolrClient`, which represents the Solr instance you want to use. Then send `SolrRequests` or `SolrQuerys` and get back SolrResponses.

`SolrClient` is abstract, so to connect to a remote Solr instance, you'll actually create an instance of either `HttpSolrClient`, or `CloudSolrClient` . Both communicate with Solr via HTTP, the difference is that `HttpSolrClient` is configured using an explicit Solr URL, while `CloudSolrClient` is configured using the zkHost String for a SolrCloud cluster.

Single node Solr client

```
String urlString = "http://localhost:8983/solr/techproducts";
SolrClient solr = new HttpSolrClient.Builder(urlString).build();
```

SolrCloud client

```
String zkHostString = "zkServerA:2181,zkServerB:2181,zkServerC:2181/solr";
SolrClient solr = new CloudSolrClient.Builder().withZkHost(zkHostString).build();
```

Once you have a `SolrClient`, 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-x.y.z.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>x.y.z</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 response format. If you are trying to mix Solr and SolrJ versions where one is version 1.x and the other is 3.x or later, then you MUST use the XML response parser. The binary format changed in 3.x, and the two javabin versions are entirely incompatible. The following code will make this change:

```
solr.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 query string:

```
SolrQuery query = new SolrQuery();
query.setQuery(mQueryString);
```

To choose a different request handler, there is a specific method available in SolrJ version 4.0 and later:

```
query.setRequestHandler("/spellCheckCompRH");
```

You can also set arbitrary parameters on the query object. The first two code lines below are equivalent to each other, and the third shows how to use an arbitrary parameter `q` to set the query string:

```
query.set("fl", "category,title,price");
query.setFields("category", "title", "price");
query.set("q", "category:books");
```

Once you have your `SolrQuery` set up, submit it with `query()`:

```
QueryResponse response = solr.query(query);
```

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 `SolrClient`'s `add()` method. This example assumes that the SolrClient object called 'solr' is already created based on the examples shown earlier.

```
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 binary format instead of the default XML format. Use the following code to upload using binary format, which is the same format SolrJ uses to fetch results. If you are trying to mix Solr and SolrJ versions where one is version 1.x and the other is 3.x or later, then you MUST stick with the XML request writer. The binary format changed in 3.x, and the two javabin versions are entirely incompatible.

```
solr.setRequestWriter(new BinaryRequestWriter());
```

Using the ConcurrentUpdateSolrClient

When implementing java applications that will be bulk loading a lot of documents at once, `ConcurrentUpdateSolrClient` is an alternative to consider instead of using `HttpSolrClient`. The `ConcurrentUpdateSolrClient` buffers all added documents and writes them into open HTTP connections. This class is thread safe. Although any SolrClient request can be made with this implementation, it is only recommended to use the `ConcurrentUpdateSolrClient` for `/update` requests.

EmbeddedSolrServer

The `EmbeddedSolrServer` class provides an implementation of the `SolrClient` 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.

Using Solr From Ruby

Solr has an optional Ruby response format that extends its JSON output in the following ways to allow the response to be safely eval'd 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 not used... data is written as raw UTF-8
- nil used for null
- => used as the key/value separator in maps

Here's an example Ruby response from Solr, for http://localhost:8983/solr/techproducts/select?q=iPod&wt=ruby&indent=on (with Solr launching using `bin/solr start -e techproducts`):

```
{
  'responseHeader'=>{
    'status'=>0,
    'QTime'=>0,
    'params'=>{
      'q'=>'iPod',
      'indent'=>'on',
      'wt'=>'ruby'}},
  'response'=>{'numFound'=>3,'start'=>0,'docs'=>[
      {
        'id'=>'IW-02',
```

```
        'name'=>'iPod & iPod Mini USB 2.0 Cable',
        'manu'=>'Belkin',
        'manu_id_s'=>'belkin',
        'cat'=>['electronics',
          'connector'],
        'features'=>['car power adapter for iPod, white'],
        'weight'=>2.0,
        'price'=>11.5,
        'price_c'=>'11.50,USD',
        'popularity'=>1,
        'inStock'=>false,
        'store'=>'37.7752,-122.4232',
        'manufacturedate_dt'=>'2006-02-14T23:55:59Z',
        '_version_'=>1491038048794705920},
      {
        'id'=>'F8V7067-APL-KIT',
        'name'=>'Belkin Mobile Power Cord for iPod w/ Dock',
        'manu'=>'Belkin',
        'manu_id_s'=>'belkin',
        'cat'=>['electronics',
          'connector'],
        'features'=>['car power adapter, white'],
        'weight'=>4.0,
        'price'=>19.95,
        'price_c'=>'19.95,USD',
        'popularity'=>1,
        'inStock'=>false,
        'store'=>'45.18014,-93.87741',
        'manufacturedate_dt'=>'2005-08-01T16:30:25Z',
        '_version_'=>1491038048792608768},
      {
        'id'=>'MA147LL/A',
        'name'=>'Apple 60 GB iPod with Video Playback Black',
        'manu'=>'Apple Computer Inc.',
        'manu_id_s'=>'apple',
        'cat'=>['electronics',
          'music'],
        'features'=>['iTunes, Podcasts, Audiobooks',
          'Stores up to 15,000 songs, 25,000 photos, or 150 hours of video',
          '2.5-inch, 320x240 color TFT LCD display with LED backlight',
          'Up to 20 hours of battery life',
          'Plays AAC, MP3, WAV, AIFF, Audible, Apple Lossless, H.264 video',
          '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'],
        'includes'=>'earbud headphones, USB cable',
        'weight'=>5.5,
        'price'=>399.0,
        'price_c'=>'399.00,USD',
        'popularity'=>10,
        'inStock'=>true,
        'store'=>'37.7752,-100.0232',
```

```
        'manufacturedate_dt'=>'2005-10-12T08:00:00Z',
        '_version_'=>1491038048799948800}]
  }}
```

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)
http_response = h.get('/solr/techproducts/select?q=iPod&wt=ruby')
rsp = eval(http_response.body)

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'] }
```

For simple interactions with Solr, this may be all you need! If you are building complex interactions with Solr, then consider the libraries mentioned at https://wiki.apache.org/solr/Ruby%20Response%20Format

Major Changes from Solr 5 to Solr 6

There are some major changes in Solr 6 to consider before starting to migrate your configurations and indexes. There are many hundreds of changes, so a thorough review of the Upgrading Solr section as well as the CHANGES.txt file in your Solr instance will help you plan your migration to Solr 6. This section attempts to highlight some of the major changes you should be aware of.

Topics discussed in this section:

- Highlights of New Features in Solr 6
- Java 8 Required
- Index Format Changes
- Managed Schema is now the Default
- Default Similarity Changes
- Replica & Shard Delete Command Changes
- facet.date.* Parameters Removed

Highlights of New Features in Solr 6

Some of the major improvements in Solr 6 include:

Streaming Expressions

Introduced in Solr 5, Streaming Expressions allow querying Solr and getting results as a stream of data, sorted and aggregated as requested.

Several new expression types have been added in Solr 6:

- Parallel expressions using a MapReduce-like shuffling for faster throughput of high-cardinality fields.
- Daemon expressions to support continuous push or pull streaming.
- Advanced parallel relational algebra like distributed joins, intersections, unions and complements.
- Publish/Subscribe messaging.
- JDBC connections to pull data from other systems and join with documents in the Solr index.

Parallel SQL Interface

Built on streaming expressions, new in Solr 6 is a Parallel SQL interface to be able to send SQL queries to Solr. SQL statements are compiled to streaming expressions on the fly, providing the full range of aggregations available to streaming expression requests. A JDBC driver is included, which allows using SQL clients and database visualization tools to query your Solr index and import data to other systems.

Cross Data Center Replication

Replication across data centers is now possible with Cross Data Center Replication. Using an active-passive model, a SolrCloud cluster can be replicated to another data center, and monitored with a new API.

Graph Query Parser

A new `graph` query parser makes it possible to to graph traversal queries of Directed (Cyclic) Graphs modelled using Solr documents.

DocValues

Most non-text field types in the Solr sample configsets now default to using DocValues.

Java 8 Required

The minimum supported version of Java for Solr 6 (and the SolrJ client libraries) is now Java 8.

Index Format Changes

Solr 6 has no support for reading Lucene/Solr 4.x and earlier indexes. Be sure to run the Lucene `IndexUpgrader` included with Solr 5.5 if you might still have old 4x formatted segments in your index. Alternatively: fully optimize your index with Solr 5.5 to make sure it consists only of one up-to-date index segment.

Managed Schema is now the Default

Solr's default behavior when a `solrconfig.xml` does not explicitly define a `<schemaFactory/>` is now dependent on the `luceneMatchVersion` specified in that `solrconfig.xml`. When `luceneMatchVersion < 6.0`, `ClassicIndexSchemaFactory` will continue to be used for back compatibility, otherwise an instance of `ManagedIndexSchemaFactory` will be used.

The most notable impacts of this change are:

- Existing `solrconfig.xml` files that are modified to use `luceneMatchVersion >= 6.0`, but do *not* have an explicitly configured `ClassicIndexSchemaFactory`, will have their `schema.xml` file automatically upgraded to a `managed-schema` file.
- Schema modifications via the Schema API will now be enabled by default.

Please review the Schema Factory Definition in SolrConfig section for more details.

Default Similarity Changes

Solr's default behavior when a Schema does not explicitly define a global `<similarity/>` is now dependent on the `luceneMatchVersion` specified in the `solrconfig.xml`. When `luceneMatchVersion < 6.0`, an instance of `ClassicSimilarityFactory` will be used, otherwise an instance of `SchemaSimlarityFactory` will be used. Most notably this change means that users can take advantage of per Field Type similarity declarations, with out needing to also explicitly declare a global usage of `SchemaSimlarityFactory`.

Regardless of whether it is explicitly declared, or used as an implicit global default, `SchemaSimlarityFactory`'s implicit behavior when a Field Types do not declare an explicit `<similarity />` has also been changed to depend on the the `luceneMatchVersion`. When `luceneMatchVersion < 6.0`, an instance of `ClassicSimilarity` will be used, otherwise an instance of `BM25Simlarity` will be used. A `defaultSimFromFieldType` init option may be specified on the `SchemaSimilarityFactory` declaration to change this behavior. Please review the `SchemaSimlarityFactory` javadocs for more details

Replica & Shard Delete Command Changes

The DELETESHARD and DELETEREPLICA now default to deleting the instance directory, data directory, and

index directory for any replica they delete. Please review the Collection API documentation for details on new request parameters to prevent this behavior if you wish to keep all data on disk when using these commands

`facet.date.*` Parameters Removed

The `facet.date` parameter (and associated `facet.date.*` parameters) that were deprecated in Solr 3.x have been removed completely. If you have not yet switched to using the equivalent `facet.range` functionality you must do so now before upgrading.

Upgrading a Solr Cluster

This page covers how to upgrade an existing Solr cluster that was installed using the service installation scripts .

⚠ The steps outlined on this page assume you use the default service name of "`solr`". If you use an alternate service name or Solr installation directory, some of the paths and commands mentioned below will have to be modified accordingly.

- Planning Your Upgrade
- Upgrade Process
  - Step 1: Stop Solr
  - Step 2: Install Solr as a Service
  - Step 3: Set Environment Variable Overrides
  - Step 4: Start Solr
  - Step 5: Run Healthcheck

Planning Your Upgrade

Here is a checklist of things you need to prepare before starting the upgrade process:

1. Examine the Upgrading Solr page to determine if any behavior changes in the new version of Solr will affect your installation.
2. If not using replication (ie: collections with replicationFactor > 1), then you should make a backup of each collection. If all of your collections use replication, then you don't technically need to make a backup since you will be upgrading and verifying each node individually.
3. Determine which Solr node is currently hosting the Overseer leader process in SolrCloud, as you should upgrade this node last. To determine the Overseer, use the Overseer Status API, see: Collections API.
4. Plan to perform your upgrade during a system maintenance window if possible. You'll be doing a rolling restart of your cluster (each node, one-by-one), but we still recommend doing the upgrade when system usage is minimal.
5. Verify the cluster is currently healthy and all replicas are active, as you should not perform an upgrade on a degraded cluster.
6. Re-build and test all custom server-side components against the new Solr JAR files.
7. Determine the values of the following variables that are used by the Solr start scripts:
   - `ZK_HOST`: The ZooKeeper connection string your current SolrCloud nodes use to connect to ZooKeeper; this value will be the same for all nodes in the cluster.
   - `SOLR_HOST`: The hostname each Solr node used to register with ZooKeeper when joining the SolrCloud cluster; this value will be used to set the **host** Java system property when starting the new Solr process.
   - `SOLR_PORT`: The port each Solr node is listening on, such as 8983.
   - `SOLR_HOME`: The absolute path to the Solr home directory for each Solr node; this directory must contain a `solr.xml` file. This value will be passed to the new Solr process using the `solr.solr.home` system property, see: Solr Cores and solr.xml.

   If you are upgrading from an installation of Solr 5.x or later, these values can typically be found in either `/var/solr/solr.in.sh` or `/etc/default/solr.in.sh`.

You should now be ready to upgrade your cluster. Please verify this process in a test / staging cluster before doing it in production.

Upgrade Process

The approach we recommend is to perform the upgrade of each Solr node, one-by-one. In other words, you will

need to stop a node, upgrade it to the new version of Solr, and restart it before moving on to the next node. This means that for a short period of time, there will be a mix of "Old Solr" and "New Solr" nodes running in your cluster. We also assume that you will point the new Solr node to your existing Solr home directory where the Lucene index files are managed for each collection on the node. This means that you won't need to move any index files around to perform the upgrade.

Step 1: Stop Solr

Begin by stopping the Solr node you want to upgrade. After stopping the node, if using a replication, (ie: collections with replicationFactor > 1) verify that all leaders hosted on the downed node have successfully migrated to other replicas; you can do this by visiting the Cloud panel in the Solr Admin UI. If not using replication, then any collections with shards hosted on the downed node will be temporarily off-line.

Step 2: Install Solr as a Service

Please follow the instructions to install Solr as a Service on Linux documented at Taking Solr to Production. After running the install script, the new Solr node will be running, so please stop it by doing: `sudo service solr stop`. You need to update the `/etc/default/solr.in.sh` include file in the next step to complete the upgrade process.



If you have a `/var/solr/solr.in.sh` file for your existing Solr install, running the `install_solr_service.sh` script will move this file to it's new location: `/etc/default/solr.in.sh` (see SOLR-8101 for more details)

Step 3: Set Environment Variable Overrides

Open `/etc/default/solr.in.sh` with a text editor and verify that the following variables are set correctly, or add them bottom of the include file as needed:

```
ZK_HOST=
SOLR_HOST=
SOLR_PORT=
SOLR_HOME=
```

Make sure the user you plan to own the Solr process is the owner of the `SOLR_HOME` directory. For instance, if you plan to run Solr as the "solr" user and `SOLR_HOME` is `/var/solr/data`, then you would do: `sudo chown -R solr: /var/solr/data`

Step 4: Start Solr

You are now ready to start the upgraded Solr node by doing: `sudo service solr start`. The upgraded instance will join the existing cluster because you're using the same `SOLR_HOME`, `SOLR_PORT`, and `SOLR_HOST` settings used by the old Solr node; thus, the new server will look like the old node to the running cluster. Be sure to look in `/var/solr/logs/solr.log` for errors during startup.

Step 5: Run Healthcheck

You should run the Solr **healthcheck** command for all collections that are hosted on the upgraded node before

proceeding to upgrade the next node in your cluster. For instance, if the newly upgraded node hosts a replica for the **MyDocuments** collection, then you can run the following command (replace ZK_HOST with the ZooKeeper connection string):

```
$ /opt/solr/bin/solr healthcheck -c MyDocuments -z ZK_HOST
```

Look for any problems reported about any of the replicas for the collection.

Lastly, repeat Steps 1-5 for all nodes in your cluster.

IndexUpgrader Tool

The Lucene distribution includes a tool that upgrades an index from previous Lucene versions to the current file format.

The tool can be used from command line, or it can be instantiated and executed in Java.

In a Solr distribution, the Lucene files are located in `./server/solr-webapp/webapp/WEB-INF/lib`. You will need to include the `lucene-core-<version>.jar` and `lucene-backwards-codecs-<version>.jar` on the classpath when running the tool.

```
java -cp lucene-core-6.0.0.jar:lucene-backward-codecs-6.0.0.jar
org.apache.lucene.index.IndexUpgrader [-delete-prior-commits] [-verbose]
/path/to/index
```

This tool keeps only the last commit in an index. For this reason, if the incoming index has more than one commit, the tool refuses to run by default. Specify `-delete-prior-commits` to override this, allowing the tool to delete all but the last commit.

Upgrading large indexes may take a long time. As a rule of thumb, the upgrade processes about 1 GB per minute.



This tool may reorder documents if the index was partially upgraded before execution (e.g., documents were added). If your application relies on monotonicity of document IDs (which means that the order in which the documents were added to the index is preserved), do a full forceMerge instead.

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 resources 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/resources#community.

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 operating in coordination with each other via ZooKeeper, and managed as a unit. A cluster may contain many collections. See also SolrCloud.

Collection

In Solr, one or more Documents grouped together in a single logical index using a single configuration and Schema. In SolrCloud a collection may be divided up into multiple logical shards, which may in turn be distributed across many nodes, or in a Single node Solr installation, a collection may be a single Core.

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 a Solr core 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 shards 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 and in coordination with each other for fault tolerance.

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

A single Replica for each Shard that takes charge of coordinating index updates (document additions or deletions) 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 automatically 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 and coordinating actions involving the entire cluster. It keeps track of the state of existing nodes, collections, shards, and replicas, and assigns new replicas 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 automatically elected to take it's place. See also SolrCloud.

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 Core that acts as a physical copy of a Shard in a SolrCloud Collection.

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 partition of a single Collection. Every shard consists of at least one physical Replica, but there may be multiple Replicas distributed across multiple Nodes for fault tolerance. See also SolrCloud.

SolrCloud

Umbrella term for a suite of functionality in Solr which allows managing a Cluster of Solr Nodes for scalability, fault tolerance, and high availability.

Solr Schema (managed-schema or schema.xml)

The Solr index Schema defines the fields to be indexed and the type for the field (text, integers, etc.) By default schema data can be "managed" at run time using the Schema API and is typically kept in a file named `managed-schema` which Solr modifies as needed, but a collection may be configured to use a static Schema, which is only loaded on startup from a human edited configuration file - typically named `schema.xml`. See Schema Factory Definition in SolrConfig for details.

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 Replica. This log is 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.

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.

Using copyField directives with suggester: Previous versions of this guide advocated using copyField directives to accumulate the contents on multiple fields into a single field to be used with Solr suggester components. This will not work previous to Solr 5.1; attempting to build the suggester will result in errors being reported in the logs if the field is multiValued. As a work-around, indexing clients should accumulate all of the contents into the field before sending the documents to Solr, and any fields used with the suggesters should have multiValued="false".

The ***variable*** **facet.range.gap parameter** was included in documentation even though the patch was not committed. As of yet there is no ability to specify variable gaps via a comma-separated list for facet.range.gap. Some of this functionality can be achieved by interval faceting, see SOLR-6216.

The **MaxIndexingThreads** parameter in **solrconfig.xml** is no longer supported from Solr 5.3, see LUCENE-6659