

LucidWorks Search System Configuration Guide

2.6.3 Documentation

Table of Contents

How to Use this Documentation	4
Conventions	4
Customers of LucidWorks Search on AWS or Azure	6
Getting Support & Training	7
Getting Started	8
LucidWorks Search User Interface Help	11
System Configuration Guide	12
Understanding LucidWorks Search	13
Collections and Indexes	36
Crawling Content	72
Query and Search Configuration	114
Security and User Management	192
Solr Direct Access	211
Performance Tips	213
Expanding Capacity	215
Integrating Monitoring Services	233
Glossary of Terms	251
A	251
B	251
C	251
D	252
F	252
I	253
M	253
N	253
Q	253
R	253
S	254
T	255
W	255
About LucidWorks	256

LucidWorks Search Documentation

The LucidWorks Search Documentation is organized into several guides that cover all aspects of using and implementing a search application with LucidWorks Search, whether on-premise or hosted on AWS or Azure.

Installation & Upgrade Guide

- Installing LucidWorks Search
- [System Directories and Logs](#)
- Upgrade instructions for v2.6
- Review changes from LucidWorks v2.5 to v2.6

LucidWorks REST API Reference

- Configure data sources and administer crawls
- Set system settings
- Manage fields, field types, and collections
- Example clients in C#, Perl and Python

System Configuration Guide

- [Troubleshooting crawl issues](#)
- [Alerts configuration](#)
- [Query options](#)
- Custom [fields](#), field types, and other [index customizations](#)
- [Performance considerations](#) and [system monitoring](#)
- [Distributed search and indexing](#)
- [Security options](#)

Custom Connector Guide

- Introduction to Lucid Connector Framework
- How To Create A Connector

Lucid Query Parser

- How the default query parser handles user requests
- Customization options

How to Use this Documentation

Audience and Scope

This guide is intended for search application developers and administrators who want to use LucidWorks Search to create world class search applications for their websites.

While LucidWorks Search is built on Solr, and many of its features are implementations of Solr and Lucene features, this Guide does not cover basic Solr or Lucene configuration. We do, however, point out where LucidWorks Search deviates from Solr or Lucene standard configuration practices, and have provided links to Solr and Lucene documentation where possible for further explanation if the functionality in LucidWorks Search is identical to Solr or Lucene.

One important note to remember is that LucidWorks is multi-core enabled by default, with `collection1` as the default core. This means that standard Solr paths such as `http://localhost:port/solr/*`, as shown in Solr documentation, would be `http://localhost:port/solr/collection1/*` in LucidWorks Search.

Topics covered on this page:

- [Audience and Scope](#)
- [Conventions](#)
- [Customers of LucidWorks Search on AWS or Azure](#)
- [Getting Support & Training](#)


Conventions





Paths

Server paths are described in relation to the base LucidWorks Search installation path, indicated by `$LWS_HOME`. For example, if LucidWorks Search was installed at `/var/lucidworks`, then the path to the 'app' directory shown as `$LWS_HOME/app` will be `/var/lucidworks/app` on the server.

Notes

Special notes are included throughout these pages.

Note Type	Look & Description
Information	<div> Notes with a blue background are used for information that is important for you to know.</div>

Notes	 Notes are further clarifications of important points to keep in mind while using LucidWorks.
Tip	 Notes with a green background are Helpful Tips.
Warning	 Notes with a red background are warning messages.
Cloud	 Information for LucidWorks Search in the Cloud Users Information specifically for LucidWorks Search customers on the AWS or Azure Platform.

REST API Conventions

Many of the LucidWorks Search REST APIs support several methods (such as POST, GET, PUT, DELETE) and each is documented with detailed attribute descriptions and examples of inputs and outputs. Each description includes the path to the API endpoint, parameters for input, and the attributes returned as a result of the request.

Windows users should take care when copying the examples as they assume that you are familiar with how to modify unix-based curl commands for the Windows environment.

Parameters

Several of the paths shown in the API documentation include parameters that need to be modified for your installation and specific configuration. These are indicated in *italics*.

For example, getting the details of a data source is shown as:

```
GET /api/collection/collection/datasources/id.
```

If you were using 'collection1' and data source '3', you would enter:

```
GET /api/collection/collection1/datasources/3.
```

Server Addresses

The LucidWorks Search REST API uses the [Core component](#), installed at <http://localhost:8888/> by default in LucidWorks Search. Many examples in this Guide use this as the server location. If you have installed LucidWorks Search locally, and you changed this location on install, be sure to change the destination of your API requests accordingly.

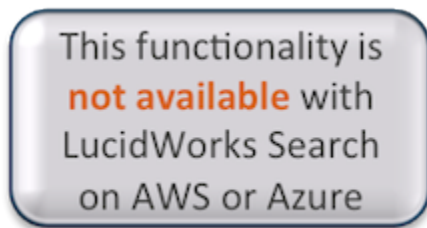
Customers hosted on AWS or Azure should see the section for [Customers of LucidWorks Search on AWS or Azure](#) below.

Customers of LucidWorks Search on AWS or Azure

All of the preceding information on this page applies to customers who have LucidWorks Search hosted on either AWS or Azure Platforms, with a few small exception which are detailed below.

Configuration Options

Certain configuration options are available with on-premise installations only (such as installation options, manual configuration file changes, etc.). The following panel will appear on any page or section that does not apply or is not available for LucidWorks Search on the AWS or Azure platforms:



API Conventions for LucidWorks Search on AWS or Azure

Nearly all of the documented REST APIs will work for customers on AWS or Azure, but the example API calls must be modified to include either the Access Key or the API Key and used as authentication credentials. Customers are being transitioned from a simple Access Key to a more secure Basic authentication system that requires a unique API Key.

1. Customers who only have an Access Key can see the key on the My Search Server page and the main Collections Overview page of your instance (click the REST API button above the usage graphs). Example URLs for API calls used in this documentation would then be changed from <http://localhost:8989/api/...> to <http://access.lucidworks.io/<access key>/api/...>. This access key is specific to your instance and should be treated as securely as possible to prevent unauthorized access via the APIs to your system.
2. Customers with Basic authentication have instances which use an URL with "https://s-XXXXXXXXX.lucidworks.io/" where XXXXXXXXX is 8 characters (letters or numbers). So, if your instance URL is "https://s-9sdf10b.lucidworks.io/" you would use that in place of any example API calls that used "http://localhost:8888". For example, this call to get all collections:

```
curl 'http://localhost:8888/api/collections'
```

would be changed to:

```
curl -u 'API_Key:password' 'https://s-9sdff10b.lucidworks.io/api/collections'
```

The API_Key can be found by logging in to your LucidWorks Search instance, and clicking "My Account" at the upper right of the screen. Click "API Access" on the left to view the API key. The password is 'x' by default. There is not currently a way to change the default password. You should take care not to expose this key when posting to our forums, as that information could be seen by other LucidWorks Search customers.

For users on LucidWorks Search for Windows Azure, the above URL would be: 'https://s-9sdff10b.azure.lucidworks.io/api/collections'.

Getting Support & Training

There are several options to get answers to questions besides this documentation:

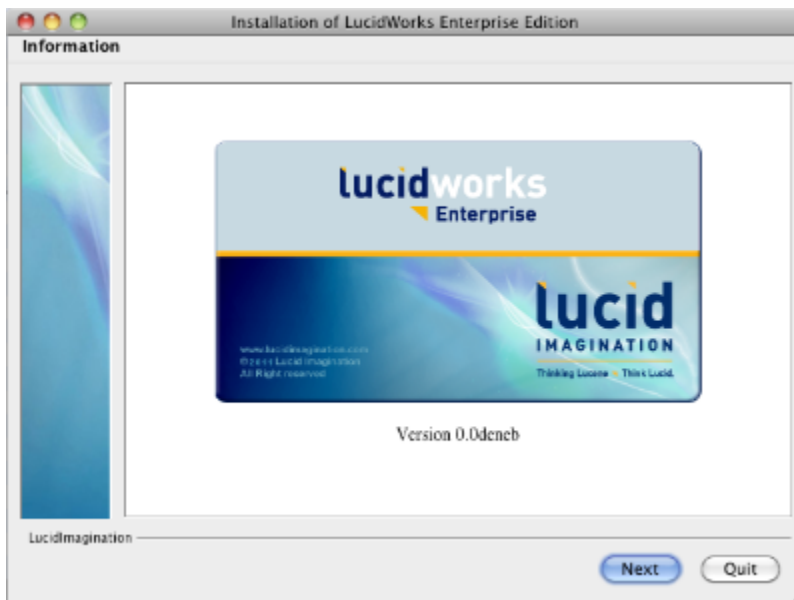
- The [LucidWorks Search Forum](#) is a place to ask questions and share information about your implementation.
- The [LucidWorks Search KnowledgeBase](#) has articles written by our support and consulting staff around common issues and questions.
- [Training Videos](#) produced by the LucidWorks training team.
- Premium support is also available, providing access to a help desk ticketing system. For more information see [Lucene/Solr Support](#).

Getting Started

The steps to get started with LucidWorks Search are not very different from getting started with any new search platform. One needs to consider the nature of the documents to be indexed, how users expect to find them, and how results will be presented to users. This section outlines those activities and points to parts of documentation to help you understand how to accomplish the necessary tasks for a successful search application.

If you are new to search applications, these sections may be helpful:

- [How Search Engines Work](#)
- [Indexing Documents](#)
- [Overview of Crawling](#)



The obvious first step is to install the application (if you are using LucidWorks Search On-Premise; LucidWorks Search on AWS or Azure, of course, is already installed).

- Installation

In general, LucidWorks Search provides two modes of interacting with the system: the Admin UI or the REST API. When just starting out, it's easier to use the Admin UI, but when developing your search application, you may want to use the API, depending on your needs. LucidWorks Search is split into three components, and it's worth getting a sense for what each one does before diving too deep into application development.

- [Working With LucidWorks Search Components](#)

Before any user can send queries to your search applications, you need to index data. LucidWorks Search requires configuring data sources for each content repository that will be added to the index and several types of repositories are supported. These can be created via the Admin UI or with the REST API.

- [Creating Data Sources with the Admin UI](#)
- [Creating Data Sources with the REST API](#)

New in v2.6 is a feature to get content into the system quickly. Called "Quick Start", it can be accessed from the UI Landing Page found at <http://localhost:8989> (be sure to adjust the host and port to the LWE-UI component as needed).

When first starting out, it's best to use a small set of documents and test that they are being indexed according to the needs of your users. The built-in Search UI was designed to be used during implementation. Queries can also be sent directly to Solr using the standard Solr syntax.

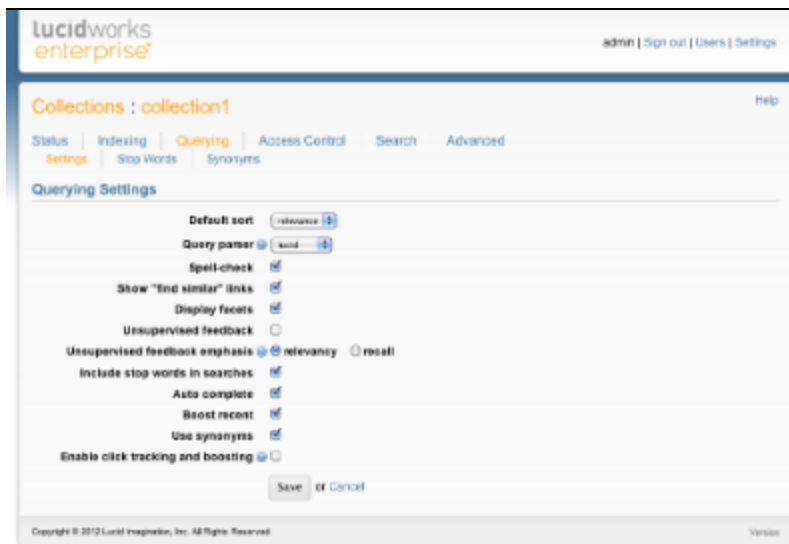
- [Using the Search UI](#)
- [Getting Search Results](#)
- [Query and Response Examples](#)

Once you see the results of initial crawls, you may realize that some of your documents don't appear as expected, or facets important to you are not appearing as you'd like.

Raw documents are broken up into various fields during the crawling and indexing processes, and the fields contained in your documents may vary from the default fields provided by LucidWorks Search through a file called `schema.xml`. While we've tried to anticipate the needs of most customers, you may find tweaks are required for your content.

In addition, LucidWorks Search provides the ability to separate indexed content into collections, that each have their own field definitions, data sources, synonym lists, activity schedules, query settings and other configurations. It's worth considering if you need to break up your content in this way, and create new collections as needed.

- [Understanding Collections](#)
- [Creating Collections with the Admin UI](#)
- [Creating Collections with the REST API](#)
- [Customizing the Field Schema](#)
- [Managing Fields with the Admin UI](#)
- [Managing Fields with the REST API](#)



Once the content is being indexed as you expect, you can modify the way user queries are handled and how results are shown to users. There are many features available, such as synonyms, auto-complete, alerting users of new results, boosting documents based on user clicks among other features.

- [Synonyms](#)
- [Stop Words](#)
- [Using User Clicks to Boost Results](#)
- [Modifying Query Settings with the Admin UI](#)
- [Modifying Query Settings with the REST API](#)
- [Lucid Query Parser Guide](#)
- [Spell Check](#)
- [Auto-Complete](#)
- [User Alerts](#)

Before going live with your search application, you'll want to consider user authentication and system security issues. LucidWorks can integrate with LDAP and supports SSL. Additionally, Access Control List information from Windows Shares can be incorporated to restrict result sets to only those documents users are allowed to see. You may also want to integrate with a JMX client, Zabbix or Nagios to monitor system performance.

- [LDAP Integration](#)
- [Restricting Access to Content](#)
- [Enabling SSL](#)
- [Securing LucidWorks](#)
- [Integrating Monitoring Services](#)

Finally, those using (or hoping to use) the SolrCloud features of LucidWorks Search will want to review the section on [Using SolrCloud in LucidWorks](#).

LucidWorks Search User Interface Help

Help for the LucidWorks Search User Interface is located at
<http://docs.lucidworks.com/display/help>.

System Configuration Guide

The System Configuration Guide provides detailed information about many of the features included with LucidWorks Search. It describes the layout of a LucidWorks Search installation and how to work with many of the configuration options included with the system. It contains the following sections:

- [Understanding LucidWorks Search](#): Introduction, location of logs, working with components
- [Collections and Indexes](#): Setting up collections, designing the index structure
- [Crawling Content](#): Crawling content of different filetypes and in different repositories
- [Query and Search Configuration](#): Configuring the user experience and how to get search results to your application
- [Security and User Management](#): SSL communication between components and user authentication
- [Solr Direct Access](#): Using Solr
- [Performance Tips](#): How to judge performance and strategies for improvement
- [Expanding Capacity](#): SolrCloud, index replication and distributed search
- [Integrating Monitoring Services](#): Using JMX, MBeans, and integrating with Zabbix or Nagios



Information for LucidWorks Search in the Cloud Users

While nearly all of the features described in this section are available to LucidWorks Search customers hosted on AWS or Azure, some of the advanced configuration options are not. When editing a setting requires direct access to a configuration file, instead of accessing the setting via the UI or an API, contact your support representative for information about how you might tweak that setting for your needs.

Understanding LucidWorks Search

This section covers the architecture of LucidWorks Search and nitty-gritty details like where log files and important directories can be found.

We also cover some introductory material: if you're not familiar with search engines, there's a section [How Search Engines Work](#) and we continue that with some more information about [How LucidWorks Search Works](#).

Then we get into the details with these sections:

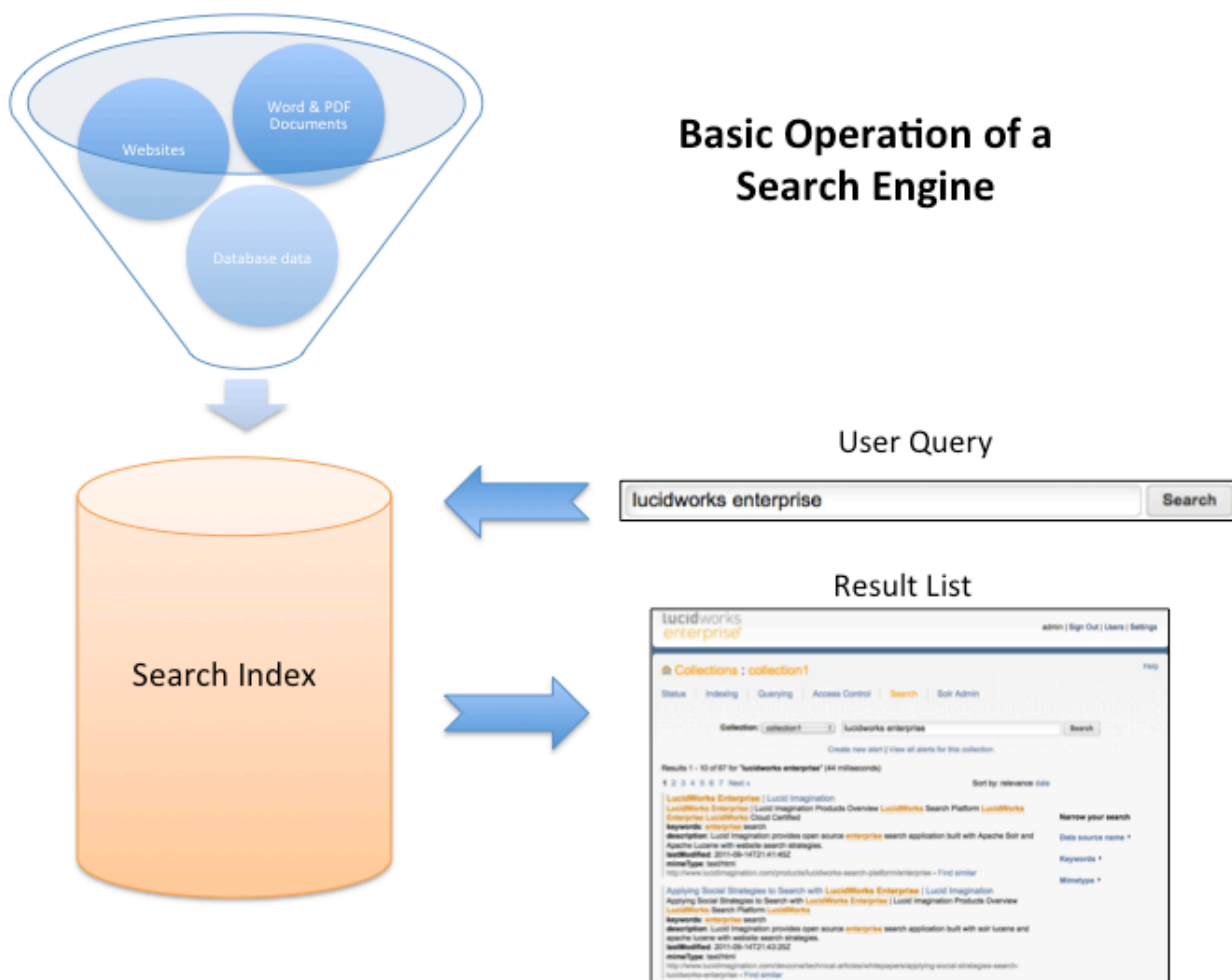
- [Working With LucidWorks Search Components](#)
- [System Directories and Logs](#)
- [Starting and Stopping LucidWorks Search](#)
- [Configuring Default Settings](#)
- [LucidWorks System Usage Monitor](#)

How Search Engines Work

In its simplest form, a search engine is an application that enables a user to query a data set and get a list of documents in response. Most people are familiar with search engines that search the internet, but search engines are also built for more specific purposes. Enterprise documents or websites are not available to the public at large, so they can't be searched with internet search engines such as Google or Yahoo. An organization may have an online store and wish to customize their site to allow customers to find products.

In LucidWorks Search, each unit of text to be searched is a "document", whether it is an article, a website, a product description, or a phone book entry. In an enterprise environment, the administrator determines which of these documents make up the data set to be searched.

This graphic shows the basic operation of a search engine:



Indexing

For a user to search a set of documents, the search engine needs to know what is in them. The process a search engine uses to find out what is in a document is called "[indexing](#)". Essentially, an administrator tells the search engine where to find the document or documents, or feeds them to the search engine by way of an uploading process. The search engine then creates an index of all the words it finds, along with a pointer to the document in which it found them. Most information within documents is organized into "[fields](#)." Fields contain information that serves a specific, important purpose in the document, such as Title, Author, or Creation Date. Good search engines are able to identify these fields and create an index for each one.

Once the search engine creates an index, lots of interesting features can be added to aid users in their search experience, such as a spelling checker, automatic query completion, faceting of results, and "find similar" functionality.

Searching

Once the search engine has created an index of available content, it is ready to accept a search. This happens when the user enters a keyword or phrase, and the search engine compares that keyword or phrase against the index, returning pointers to any documents that are associated with them.

Of course, people are surprisingly different in the way in which they approach a topic, so search engines need to take these variations into account. The goal of a search engine is to match words entered by a user to words found in a document, so one technique it uses is to "normalize" both the user's query and terms that have been indexed as much as possible to find the best possible match, similar to the way in which you might convert both a target string and the text you are matching to uppercase in order to eliminate case-sensitivity.

Full-text Searching and Challenges

Several inherent challenges complicate full-text search. First, there is currently no way to guarantee the searcher will find the "best" results because there is often no agreement on what the "best" result is for a particular search. That's because evaluating results can be very subjective. Also, users generally enter only a few terms into a search engine, and there is no way for the search system to understand the user's intention for a search. In fact, if the user is doing an initial exploration of a topic area, the user may not even be aware of his or her intention.

A system that understands natural language (that is, the way people speak or write) perfectly is usually considered the ultimate goal in search engine technology, in that it would do as good a job as a person in finding answers. But even that is not perfect, as variations in human communication and comprehension mean that even a person is not guaranteed to find the "right" answer, especially in situations where there may not even be a single "right" answer for a particular question.

Some search engines, such as LucidWorks Search, are built with features that try to solve, or at least mitigate, these challenges. This System Configuration Guide will introduce you to many of these features and describe how to configure them.

How LucidWorks Search Works

Like any other search engine, LucidWorks Search works by indexing several kinds of documents and providing a way for a user to search them. It uses Lucene and Solr to handle the core indexing and query processing tasks, and leverages the latest advancements in those projects. LucidWorks also builds on the work of the open-source community by adding [crawling features](#), a robust REST API, an easy-to-use administration interface, and other features.

The Apache Solr/Lucene core provides the indexing and searching functionality on which LucidWorks is built. As an application developer using LucidWorks Search, you can access this functionality in the same way that you access a traditional Solr installation. This includes field definition, document analysis, faceting, and basic query interpretation. Customers with LucidWorks Search installed on their own servers can work with the Apache Solr/Lucene core directly if they choose. Customers who use LucidWorks Search on AWS or Azure access much of the same functionality through the [Admin UI](#).

On top of the Apache Solr/Lucene core is LucidWorks Search, which provides programmatic and GUI access to features that are normally difficult to work with directly, such as field definition or data source creation and scheduling.

- The [LucidWorks Search Admin User Interface](#) provides configuration and management tools for almost every feature of LucidWorks, including document acquisition, security, and field definitions.
- The REST API provides programmatic access to almost all configuration and management functions within LucidWorks.

Most of the functionality provided by LucidWorks comes from the [LWE-Core and LWE-Connectors components](#), which manage all of these processes and features so administrators can concentrate on building and managing their own applications rather than the underlying search engine.

Related Topics

- [Working With LucidWorks Search Components](#)
- [Indexing Documents](#)
- [Getting Started](#)

Working With LucidWorks Search Components

LucidWorks Search has three main components that can each be run together on a single server or deployed on separate servers if desired. While LucidWorks Search customers on AWS or Azure will not often need to interact with these components, an understanding of how they work is helpful for a deeper understanding of the system as a whole.

- [About the Components](#)
 - [LWE-Core](#)
 - [LWE-UI](#)
 - [LWE-Connectors](#)
 - [Default Installation URLs](#)
- [Configuring the Components](#)
- [Related Topics](#)

About the Components

Each component is a single JVM process. The system properties for each JVM can be modified with the `master.conf` file found in the `$LWS_HOME/conf` directory.

LWE-Core

The LucidWorks Search Core component is the main engine of the application. It contains the search index, the index definitions, the query parser, the embedded [Solr](#) application and Lucene libraries, as well as serves the REST API (with the exception of Alerts).

LWE-UI

The UI component includes all web-based graphical interfaces for administering the application, a sample search interface, Relevancy Workbench and the enterprise alerts feature.

Through the Admin UI, you can modify index fields, configure data sources for content collection, define aspects of the search experience, and monitor system performance.

The Search UI provides a front-end for users to submit queries to LucidWorks Search and review results. It is not intended as a production-grade user interface, rather as a sample interface to use while configuring and testing the system.

Relevancy Workbench is a tool to model possible changes to how user query terms are interpreted in order to improve relevancy. More information about this tool is available at [Relevance Workbench](#).

Enterprise Alerts provide a way for users of the front-end Search UI to save searches and receive notifications when new results match their query terms. There is a [user interface](#) piece with forms and screens for users to configure and review their alerts, as well as a REST API for programmatic access to the Alerts features.

LWE-Connectors

The Connectors component performs all the crawler functions, which include crawling data sources on demand or at a specific schedule, maintaining a crawl history (as applicable; each crawler varies in their behavior), and saving data source configuration information for use by the crawlers. The Connectors component also manages the [LucidWorks Logs](#) crawler.

Default Installation URLs

This guide will refer to example URLs that will reference the default installation URLs for each component. These defaults are:

Component	Default URL	Web Interfaces
LWE-Core	http://127.0.0.1:8888/	This URL is used as the base for accessing most of the REST APIs, and also for accessing Solr Admin UI at http://127.0.0.1:8888/solr . .
LWE-UI	http://127.0.0.1:8989/	There are multiple front-ends at this URL. This base URL will access the Landing Page, which will provide access to the Quick Start, the LucidWorks Search Admin UI, Relevancy Workbench, and also a link to the Solr Admin UI.
LWE-Connectors	http://127.0.0.1:8765/	There is no web front-end at this URL, it is used by the LWE-Core and LWE-UI components to communicate with the Connectors component.

These URLs are used by the installer for two purposes:

1. When the various components communicate with each other, or link to one another, they specify which URL will be used.
2. If the "Enable" check box is selected for a component when using the installer, then that component will be run locally, using the port specified in the URL.



The default LucidWorks start scripts start all components at the same time. However, it is possible to restart or stop a single component. See the section [Starting and Stopping LucidWorks Search](#) for details.

[Back to Top](#)

Configuring the Components

If all components are run on the same machine, they must be defined with different ports. They can also be configured to run on different servers.

There are three possible ways to configure the components:

1. All components run on the same machine and they are started and stopped together. This is the default for the standalone installer, which automatically prompts for default ports that are different for each component. To use this mode, you only need to run the installer once and follow through the process completely.
2. All components run on the same machine but they are started and stopped separately. This would require running the installer three times on the same machine. See [Installing Components on Different Servers](#) for detailed instructions on how to do this.
3. Each component is on a different machine and started and stopped separately. This requires running the installer on each machine. See [Installing Components on Different Servers](#) below for detailed instructions on how to do this.

[Back to Top](#)

Related Topics

- [Expanding Capacity](#)

System Directories and Logs

This functionality is
not available with
LucidWorks Search
on AWS or Azure

There are several important directories in the LucidWorks Search installation. System activities are recorded in several log files. Knowing where files and logs are located will make system configuration and troubleshooting easier.

- [Locating Files and Directories](#)
 - [Configuring LucidWorks Search Directories](#)
 - [Temporary Files](#)
- [System Logs](#)
 - [Log Properties](#)
- [LucidWorksLogs Collection](#)
- [Related Topics](#)

Locating Files and Directories

The following table shows the default location of some directories that may be needed to effectively work with LucidWorks Search. These paths are all relative to the LucidWorks Search installation path (referred to as `$LWS_HOME`) which is specified during installation.

What	Path
Configuration Files	<code>\$LWS_HOME/conf/</code>
Documentation	<code>\$LWS_HOME/app/docs/</code> (PDF) or http://docs.lucidworks.com (Online)
Examples	<code>\$LWS_HOME/app/examples/</code>
Jetty Libraries	<code>\$LWS_HOME/app/jetty/lib/</code>
Licenses	<code>\$LWS_HOME/app/legal/</code>
Logs	<code>\$LWS_HOME/data/logs/</code> (See below for log file list)
LucidWorks Indexes	<code>\$LWS_HOME/data/solr/cores/collection/data/</code>
LucidWorks Logs	<code>\$LWS_HOME/data/solr/cores/LucidWorksLogs/data/</code>
Solr Home	<code>\$LWS_HOME/conf/solr/</code>
Solr Configuration Files	<code>\$LWS_HOME/conf/solr/cores/collection/conf/</code>
Solr Source Code	<code>\$LWS_HOME/app/solr-src/</code>

Start/Stop Scripts	\$LWS_HOME/app/bin/
--------------------	---------------------

Editing Configuration Files on Windows

LucidWorks Search holds configuration files open after reading them, which may cause problems on Windows systems that do not allow editing open files. In this case, stop LucidWorks Search before editing files on Windows to be sure the edits are saved properly.

Configuring LucidWorks Search Directories

After you have installed LucidWorks Search, you can configure the location of the `app`, `conf`, `data`, and `logs` directories by passing these parameters to the start script (`start.sh` or `start.bat`):

- `-lwe_app_dir`
- `-lwe_conf_dir`
- `-lwe_data_dir`
- `-lwe_log_dir`

For example, to change the location of the `data` directory, pass the following parameter to your start script:

```
start.sh -lwe_data_dir /var/data
```

See the section on [Starting and Stopping LucidWorks Search](#) for more information about the start scripts.

Temporary Files

By default, LucidWorks Search uses standard system directories (as detected by the JVM) for creating temporary files. This can be changed by adding a system property to the `master.conf` for `java.io.tmpdir` in the section that controls each JVM for the system. For example, to change the location of temporary files for the LucidWorks Core component, you would follow these steps:

1. Shut down LucidWorks using the instructions found in the section on [Starting and Stopping LucidWorks Search](#).
2. Open `master.conf` with a text editor (found in `$LWS_HOME/conf`).
3. Find the section for `lwecore.jvm.params` and add `-Djava.io.tmpdir=/tmp/files/`.
4. Start LucidWorks.

The directory chosen as the location for temporary files should exist before starting LucidWorks Search, and must be writable by the user running LucidWorks.

[Back to Top](#)

System Logs

LucidWorks Search records system activities to rolling log files located in the `$LWS_HOME/data/logs` directory of the installation by default. The table below describes the main purpose of the various log files.

Log Name	Name Pattern	Function
Connector component log	<code>connectors.<YYYY_MM_DD>.log</code>	Connectors component operations, including the output of all crawling operations.
Connector request log	<code>connectors.request.<YYYY_MM_DD>.log</code>	Requests to the connectors component. These usually come from the Core component.
Core component log	<code>core.<YYYY_MM_DD>.log</code>	LucidWorks Core component operations, such as indexing.
Core request log	<code>core.request.<YYYY_MM_DD>.log</code>	Requests to the core component. These could come from either the Connectors or the UI component.
Core standard error log	<code>core-stderr.log</code>	Errors from Jetty startup (if any).
Core standard output log	<code>core-stdout.log</code>	Messages from Jetty startup (if any).
UI component log	<code>ui.<YYYY_MM_DD>.log</code>	Information from the Rails application, which runs the Search, Admin and Alerts components.
UI request log	<code>ui.request.<YYYY_MM_DD>.log</code>	Requests to the UI component.
Ruby standard error log	<code>ruby-stderr.log</code>	Errors from Ruby startup (if any).
Ruby standard output log	<code>ruby-stdout.log</code>	Messages from Ruby startup (if any).
Click log	<code>click-<collectionName>.log</code>	User click data, for use in relevance boosting (if enabled).

SharePoint crawl log	google_connectors.feed.log	SharePoint crawling operations. Note, this file can also include a number in the name, such as google_connectors.feed0.log, etc.
-------------------------	----------------------------	---

Log files are available through the Admin UI, by going to the Server Logs page for a collection and clicking the link at the bottom of the page. If for some reason the Admin UI is not available, log files can be downloaded with a curl command to the Core component such as:

```
curl http://localhost:8888/logs/<log_file_name>
```

Note, however, if the LucidWorks Search Core component is down, that curl command will not work.

Log Properties

The LucidWorks Search Core log is configured by the `$LWS_HOME/conf/log4j-core.xml` properties file. The default is to create a distinct log per date (server time).

The LucidWorks Search UI log is configured by the `$LWS_HOME/conf/log4j-ui.xml` properties file. The default is to create a distinct log per date (server time).

The LucidWorks Search Connector log is configured by the `$LWS_HOME/conf/log4j-connectors.xml` properties file. The default is to create a distinct log per date (server time).



The LucidWorks Search Connectors log includes information about crawl activities such as attempts to access a file or URL and the results of those attempts. By default, the log does not record the collection or data source associated with crawl activities. However, if you would like to record that information for later review, you can edit the `$LWS_HOME/conf/log4j-connectors.xml` file.

In the file, find the section that begins with a comment to "Use the pattern below to log additional context info...", as below:

```
<!-- Use the pattern below to log additional context info like collection and
data source name -->
<!--
  <param value="%d{ISO8601} %p %c{2} - %X %m%n" name="ConversionPattern"/>
-->
```

Uncomment `<param value="%d{ISO8601} %p %c{2} - %X %m%n" name="ConversionPattern"/>` and save the file. You should restart LucidWorks Search after making this change.

More information on how to modify log4j settings for the Core and UI log files is available at <http://logging.apache.org/log4j/1.2/manual.html>.

[Back to Top](#)

LucidWorksLogs Collection

LucidWorks Search records log files for your Solr indexes in a collection called LucidWorksLogs, which contains a pre-configured data source also called `lucidworkslogs`. You can view the data for the LucidWorksLogs collection as you would for any other collection. You can also access the log files directly in the `$LWS_HOME/data/solr/cores/LucidWorksLogs/` directory.

The LucidWorksLogs collection powers the error log and all statistics about recent query and indexing activity that is shown in the Admin UI.

The log files on a LWE-Core server are accessible via HTTP at the URL

"`http://server:port/logs`". This URL lists all files currently in the logs directory, and provides links for downloading them individually. This can be useful in situations where you do not have direct shell access to the LWE-Core machine, but would like to review the log files for troubleshooting purposes.

If you are using LucidWorks Search in SolrCloud mode or with each component installed on a different server, please see the section Log Indexing with Separated Components for details on how to make sure your logs are fully indexed.

When securing the HTTP Port of LWE-Core installation, consideration should be taken as to whether the `/logs` directory should be secured or not.



Deleting the LucidWorksLogs Collection

It is possible to delete the LucidWorksLogs collection if desired; however, this will disable the server log page within other collections, all activity graphing, and all calculations of Most Popular and Most Recent queries.

If the collection was deleted in error, or if you'd like to restore it at a later time, go to the Server log page within any collection and click **Recreate the log collection**.

It is also possible to remove the LucidWorksLogs data source from the LucidWorksLogs collection (i.e., retain the collection for possible later use, but remove the mechanism that indexes the logs). However, at the current time it will automatically be re-created and re-scheduled on server restart. If you wish to disable log crawling, you must either remove the entire LucidWorksLogs collection, or modify the LucidWorksLogs data source so that the schedule is not active (you can modify the schedule with the Data Source Schedules API or in the Schedules screen of the Admin UI).

Related Topics

-
- [Working With LucidWorks Search Components](#)
 - [Starting and Stopping LucidWorks Search](#)

[Back to Top](#)

Starting and Stopping LucidWorks Search

This functionality is
not available with
LucidWorks Search
on AWS or Azure

LucidWorks Search can be started and stopped using start and stop scripts provided with the application. These scripts are described below.

✔ Windows users who have configured LucidWorks Search to run as a service should use the Services panel in Windows to manage start and stop.

- [Starting a Standalone LucidWorks Search Instance](#)
- [Starting SolrCloud-enabled LucidWorks Search Instances](#)
 - [Passing SolrCloud parameters at Start](#)
 - [Updating `master.conf`](#)
- [Stopping LucidWorks Search \(all modes\)](#)
- [Starting or Stopping Components Separately](#)

Starting a Standalone LucidWorks Search Instance

If you did not allow the installer to start LucidWorks Search, or if shortcuts were not installed, you can still start or stop the system manually from the command line. This will start all components:

1. Open a command shell, and make sure Java 1.6 or greater is in your path.
2. Change directories to the LucidWorks installation directory, then to the `$LWS_HOME/app/bin` directory.
3. Invoke `start.sh` for UNIX/Mac/Cygwin or `start.bat` for Windows systems.

⚠ If you are using LucidWorks Search in SolrCloud mode, please refer to the section [Starting LucidWorks Search](#) in the documentation for Using SolrCloud in LucidWorks Search.

Starting SolrCloud-enabled LucidWorks Search Instances

If you are using LucidWorks Search in SolrCloud mode, you must start the application in a way that the underlying Solr instances are aware of where ZooKeeper is. If you used the LucidWorks Search installer, the required parameters have been added to the `conf/master.conf` file for each instance.

However, if you bootstrapped LucidWorks Search manually, or installed without the all of the SolrCloud installer steps, you will need to pass the required parameters on the command line. You can also manually update `conf/master.conf` file.

Passing SolrCloud parameters at Start

As long as the initial bootstrap has been completed (if not, please see [Starting LucidWorks Search](#)), the only parameter that is required on future startup is the `zkHost` parameter. This parameter points to each of the ZooKeeper instances and the root directory for the configurations that are stored in ZooKeeper. This example command starts LucidWorks Search and points to an external ZooKeeper:

```
$ ./start.sh -lwe_core_java-opts  
"-DzkHost=10.0.1.7:5001,10.0.1.9:5001,10.0.1.11:5001/lws"
```

If you are using the embedded ZooKeeper instance, then you may alternately need to start ZooKeeper while starting LucidWorks Search with the `zkRun` parameter on one of the instances. Subsequent instances would require the `zkHost` parameter to point to the instance with the running ZooKeeper. For example, to start the first instance:

```
$ ./start.sh -lwe_core_java-opts "-DzkRun"
```

Then all subsequent instances are started:

```
$ ./start.sh -lwe_core_java-opts "-DzkHost:10.0.1.7:9988"
```

Note when using the embedded ZooKeeper that the port is the LWE-Core component port + 1000.

Updating master.conf

If you don't want to have to pass the ZooKeeper parameters each time you restart, you can modify the `conf/master.conf` file for each instance. Simply add the `-DzkHost` parameters to the section JVM Settings of LWE-Core and they'll be passed to the start script. For example, here is a sample where:

```
# COMPONENT LWE-Core - LWE-Solr + LWE REST API.  
# -----  
lwecore.enabled=true  
lwecore.address=http://10.0.1.5:8888  
  
# JVM Settings for LWE-Core  
lwecore.jvm.params=-Xms512M -Xmx1024M -XX:MaxPermSize=256M -Duser.language=en  
-Duser.country=US -Duser.timezone=UTC -Dfile.encoding=UTF-8  
-Dcom.sun.management.jmxremote -DzkHost=10.0.1.7:5001,10.0.1.9:5001,10.0.1.11:5001/lws
```

If using the embedded ZooKeeper instance, the same approach can be taken to add the `-DzkRun` parameter to one instance, with `-DzkHost` being added to the other instances.

These parameters only need to be added to the LWE-Core component for each instance that runs the LWE-Core component; so if you have an instance that is only running the UI or the Connectors, the parameters don't need to be added at all.

Stopping LucidWorks Search (all modes)

To stop LucidWorks Search, use the stop scripts at the command line. This will stop all components and any running processes.

1. Open a command shell, and make sure Java 1.6 or greater is in your path.
2. Change directories to the LucidWorks installation directory, then to the `$LWS_HOME/app/bin` directory.
3. Invoke `stop.sh` for UNIX/Mac/Cygwin or `stop.bat` for Windows systems.



Restarting LucidWorks Search

To restart LucidWorks Search, first stop the servers and start them again using the processes outlined above.

Starting or Stopping Components Separately

To start or stop any of the components without starting or stopping the other components, you can use the `start.sh/start.bat` or `stop.sh/stop.bat` scripts with the `-only` parameter, followed by the component name.

- Core component: `lwe-core`
- UI component: `lwe-ui`
- Connectors component: `connectors`

For example, this would start only the connectors using the `start.sh` script:

```
start.sh -only connectors
```

Configuring Default Settings

This functionality is
not available with
LucidWorks Search
on AWS or Azure

You can configure many default settings in LucidWorks Search in the `defaults.yml` file located in the `$LWS_Home/conf/lwe-core` directory. You must [restart LucidWorks](#) after editing this file for your changes to take effect.

Some of the default settings you can configure include:

- Default crawl depth
- Default field mappings for crawlers
- Batch crawling of data sources
- Enabling or restricting data sources by crawler
- Default HTTP proxy settings

For example, to set the default crawl depth to 3 (which means that the crawler will follow links/sub-directories up to three steps away from the initial target), set `datasource.crawl_depth`: 3.

Here is an example `defaults.yml` file with comments that explain the various default settings (your default.yml file may vary):

```
file: defaults.yml
initCalled: true
location: CONF
values:
# Set to true to block index updates
  control.blockUpdates: false
# A whitespace-separated list of symbolic crawler names to enable; all crawlers are
# enabled if this list is empty
  crawlers.enabled.crawlers: ''
# Absolute path that will be used to resolve relative path of local file system crawls
  crawlers.filesystem.crawl.home: null
# Per-crawler list of enabled datasource types, whitespace-separated. All available
# types are enabled if this list is empty.
  crawlers.lucid.aperture.enabled.datasources: ''
# Per-crawler whitespace-separated list of restricted datasource types; all enabled
# types are unrestricted if this list is empty
  crawlers.lucid.aperture.restricted.datasources: ''
  crawlers.lucid.external.enabled.datasources: ''
  crawlers.lucid.external.restricted.datasources: ''
  crawlers.lucid.fs.enabled.datasources: ''
  crawlers.lucid.fs.restricted.datasources: ''
```

```
crawlers.lucid.gcm.enabled.datasources: ''
crawlers.lucid.gcm.restricted.datasources: ''
crawlers.lucid.jdbc.enabled.datasources: ''
crawlers.lucid.jdbc.restricted.datasources: ''
crawlers.lucid.logs.enabled.datasources: ''
crawlers.lucid.logs.restricted.datasources: ''
crawlers.lucid.solrxml.enabled.datasources: ''
crawlers.lucid.solrxml.restricted.datasources: ''
# Default data source bounds: choose none or tree
datasource.bounds: none
# Batch processing; caching of crawled raw content
datasource.caching: false
# Explicitly commit when crawl is finished
datasource.commit_on_finish: true
# Solr's commitWithin setting, in milliseconds
datasource.commit_within: 900000
# Default crawl depth: the number of cycles or hops from the root URL/directory. Set to
-1 for unlimited crawl depth
datasource.crawl_depth: -1
datasource.follow_links: true
# Set to true to ignore the rules defined in /robots.txt for a site
datasource.ignore_robots: false
# Perform indexing at the same time as crawling
datasource.indexing: true
# Default field mapping for Aperture-based crawlers. This is the baseline, the field
mapping for each data source can be customized.
datasource.mapping.aperture: &id001
!!com.lucid.admin.collection.datasource.FieldMapping
  datasourceField: data_source
  defaultField: null
  dynamicField: attr
  literals: {}
  mappings:
    slide-count: pageCount
    content-type: mimeType
    body: body
    slides: pageCount
    subject: subject
    plaintextmessagecontent: body
    lastmodified: lastModified
    lastmodifiedby: author
    content-encoding: characterSet
    type: null
    date: null
    creator: creator
    author: author
    title: title
    mimetype: mimeType
    created: dateCreated
    plaintextcontent: body
```

```
pagecount: pageCount
contentcreated: dateCreated
description: description
contributor: author
name: title
filelastmodified: lastModified
fullname: author
fulltext: body
messagesubject: title
last-modified: lastModified
acl: acl
keyword: keywords
contentlastmodified: lastModified
last-printed: null
links: null
url: url
batch_id: batch_id
crawl_uri: crawl_uri
filesize: fileSize
page-count: pageCount
content-length: fileSize
filename: fileName
multiVal:
  fileSize: false
  body: false
  author: true
  title: false
  acl: true
  description: false
  dateCreated: false
types:
  filesize: LONG
  lastmodified: DATE
  datecreated: DATE
  date: DATE
uniqueKey: id
# Default field mapping for crawlers that use Tika parsers
datasource.mapping.tika: *id001
# Maximum size of content to be fetched
datasource.max_bytes: 10485760
# Maximum number of documents to collect; set to -1 for unlimited documents
datasource.max_docs: -1
# The maximum number of concurrent requests processed by a data source crawl, for those
crawlers that support multi-threaded crawling.
# As of v2.1, this is only the lucid.fs crawler, which supports the Hadoop, S3 and SMB
data source types.
datasource.max_threads: 1
# Set to true to apply content parsers to the retrieved raw documents
datasource.parsing: true
# Defines the host name of an HTTP proxy server to use for web crawling; leave blank if
```

```
you are not using a proxy server
datasource.proxy_host: ''
# HTTP proxy password, if you are using an HTTP proxy server
datasource.proxy_password: ''
# proxyPort for an HTTP proxy server, if you are using one
datasource.proxy_port: -1
# Username to authenticate with HTTP proxy server
datasource.proxy_username: ''
# If true, text extracted from a compound document (one which has other embedded
documents and resources, such as emails with attachments
# or Office documents with OLE attachments, but not .zip, .jar., or similar) will be
appended to the text of the container document.
# If false, each embedded resource is treated as a separate document with a URL in the
form of the container document URL plus ! and
# the embedded document's name or identifier. If documents are treated as separate
documents (when this setting is false),
# the URL of the container document is added to the field "belongsToContainer".
datasource.tika.parsers.flatten.compound: true
# If false, documents with mime types that start with "image/" are ignored. If true,
the documents are sent to Tika for parsing,
# which may result in useful metadata being extracted from them but may also result in
a large number of fields and terms.
datasource.tika.parsers.include.images: false
# If true, and LucidWorks runs in the same JVM as Solr, then crawlers will first try
using direct calls to SolrCore for updates,
# which may result in performance improvements. If false (the default), the SolrJ API
is used for updates.
datasource.use_direct_solr: false
# If true, datasources will attempt to verify access to the remote repositories.
datasource.verify_access: true
# HTTP-specific preferences sent in HTTP headers during crawling.
http.accept.charset: utf-8,ISO-8859-1;q=0.7,*;q=0.7
http.agent.browser: Mozilla/5.0
http.agent.email: crawler at example dot com
http.agent.name: LucidWorks
# The agent.string will allow a completely custom http.agent identifier. If this is not
empty, it will be used verbatim instead of all other 'http.agent.*' settings.
http.agent.string: ''
http.agent.url: ''
http.agent.version: ''
http.crawl.delay: 2000
# Maximum number of redirections in a redirection chain.
http.max.redirects: 10
# Number of threads for HTTP crawling.
http.num.threads: 1
# Socket timeout in milliseconds.
http.timeout: 10000
# Specify the HTTP version: HTTP/1.1 if true; HTTP/1.0 if false.
http.use.http11: true
ssl.auth_require_authorization: false
```



```
ssl.auth_require_secure: false
```

Related Topics

- [Overview of Crawling](#)

LucidWorks System Usage Monitor

The LucidWorks System Usage Monitor is a voluntary program to allow LucidWorks Search users to anonymously send basic information about their system to LucidWorks. We use this information to analyze the types of systems in use by our customers and how they are used so we can improve our product. At no point does the system collect information that could identify you, your organization, the documents indexed, or the type of content indexed.

Information Collected

The System Usage Monitor collects the following information for LucidWorks Search installations:

- Operating System version and type
- Java version and type
- LucidWorks Search version and type
- Number of LucidWorks Search collections created
- Number of LucidWorks Search data sources created
- Number of LucidWorks Search documents indexed
- JVM memory free, available, and used
- Number of LucidWorks Search queries
- Number of documents added since last submission

How the System Usage Monitor Works

When Information is Sent

The System Usage Monitor sends information at each LucidWorks startup (using the `start.sh` or `start.bat` scripts) and once per week on Saturdays.

How Information is Sent

When LucidWorks Search is started, the System Usage Monitor will transmit data about your system to a server hosted by LucidWorks with two HTTP requests. The first request contains system-level information and if that is successful, the second request will send LucidWorks-specific information, as listed above.

The information is sent via an encrypted POST request to <https://heartbeat.demo.lucidworks.io>. Each request includes a unique identifier, which is anonymous and can't be used to identify the sender. The IP that sent the request is not stored with the request.

The requests are logged in the LucidWorks Search core log (`core.YYYY_MM_DD.log`). The requests will appear similar to this:

```
2012-10-23 19:05:56,618 INFO heartbeat.LucidStatsPublisher - Sending heartbeat stats:
uuid='3532f7e9-4280-4714-9e83-ea0a95fe90bd',data='{product=lwe,
current_product_version=0.0Enif, is_cloudy=false,
lwe_git_sha=7568ce8c35a394c4b987e3a17cb5e1b5ae5dac25,java_version=1.6.0_35 (Apple
Inc.), num_cpu_cores=4, os_version=Mac OS X (x86_64)}'2012-10-23 19:05:58,831 INFO
publish.MonitorRegistryMetricPoller - cache refreshed, 8 monitors matched filter,
previous age 1351019158 seconds
2012-10-23 19:05:58,865 INFO heartbeat.LucidStatsPublisher - Sending heartbeat stats:
uuid='3532f7e9-4280-4714-9e83-ea0a95fe90bd',data='{num_docs=0, num_collections=1,
num_datasources=0, jvm_memory_free=506720952, jvm_memory_max=1065025536,
jvm_memory_total=534708224,num_adds=0, num_search_requests=0}'
```

Subsequent weekly updates are sent as a single request, including only the LucidWorks Search-specific information like number of documents, number of data sources, etc.

How to Opt-In or Opt-Out

During Installation

During installation of LucidWorks Search, you will be presented with an option to opt-in to the System Usage Monitor program. This option will appear after defining the installation path for the system. With the graphical installer, the box is checked by default and un-checking the box will opt-out of the program. If using the console installer, choose '0' as a response to opt-out of the program.

Post-Installation

Opting-in to the program will insert a line at the beginning of the `$LWS_HOME/conf/master.conf` file, as so:

```
# LucidWorks System Usage Monitor (comment the next line to disable this feature)
usageStatsServerId=3532f7e9-4280-4714-9e83-ea0a95fe90bd
```

To opt-out:

1. Stop LucidWorks Search
2. Open `master.conf` found in `$LWS_HOME/conf`
3. Comment out the line containing the `usageStatsServerID` by adding a hash mark (#) at the beginning of the line
4. Start LucidWorks Search

The same process can be followed to opt-in if the service was previously disabled, by removing the hash mark instead of inserting it.

More Information

For more information, including details of our commitment to protecting the privacy of your data, please see our website at <http://www.lucidworks.com/lucidworks-system-usage-monitor>.

Collections and Indexes

This section covers how to configure LucidWorks Search for your data.

Content in LucidWorks is indexed into a collection, which can have different documents, data sources, fields, field types and settings from other collections. Before starting to work with LucidWorks Search, review the section [Working with Collections](#). Once one collection is configured as you like, it can be used as a template, as described in [Using Collection Templates](#).

Once the collections are considered, then you can think about how to configure LucidWorks Search to index your content. These sections describe the options for setting up the indexes:

- [Indexing Documents](#)
- [How Documents Map To Fields](#)
- [Customizing the Field Schema](#)
- [Reindexing Content](#)
- [Multilingual Indexing and Search](#)
- [Lucid Plural Stemming Rules](#)
- [Deleting the Index](#)

Working with Collections

A single installation of LucidWorks Search may be used to index multiple types of content, serve multiple user constituencies, or accommodate multiple overlapping security rules. It does this by supporting the creation and use of multiple "collections". A collection is a set of documents that are grouped together with the same indexing and query rules. Each collection in LucidWorks has its own index and configuration files and is logically separate from all other collections.

For those familiar with Solr, the concept of collections in LucidWorks is very similar to the concept of [cores](#) in Solr.

Default Collections

By default, each LucidWorks Search installation includes two collections out of the box: "collection1" and "LucidWorksLogs".

Collection1 is the primary collection used by LucidWorks Search to store indexes and define query settings. It can be used as-is immediately after installation to start indexing documents and using the default Search UI. However, a collection cannot be renamed once created (nor can content be moved from one collection to another without indexing it all from scratch). So, if you think you'll use multiple collections and want to name each one based on what it contains or what it will be used for, you would probably create a new collection and start from there.

The LucidWorksLogs collection is a special collection, used to index logs for easier troubleshooting. It is discussed in more detail in the section on the [LucidWorksLogs collection](#). It can be deleted at any time and recreated later, if desired.

If you want to delete collection1, you can do so after you've created at least one other standard collection, as there must always be at least one collection (not including the LucidWorksLogs collection).

A collection that has been customized can also be used as the basis for future collections; see the section on Collection Templates for more information.

Per-Collection Features

You can configure the following items for each collection individually:

- Data sources
- Fields
- Query settings
- Search UI
- Search Filters
- Schedules
- Solr Admin

After you have created additional collections, you should pay special attention to the collection name you are working with so you edit the proper configuration files or make the correct API calls. This is particularly true when using the REST API or several of the advanced configuration options discussed later in this Guide, but it also applies to the various screens of the [Admin UI](#). Modifying the wrong collection out of context may have unexpected consequences including poorly indexed content or an inconsistent search experience for users.

System-Wide Features

The following items are system-wide and can only be configured for the entire LucidWorks Search installation or instance:

- Collection definition
- Access to user interfaces
- Users
- Alerts (although these take the collection as a parameter to limit the query)

Related Topics

- Creating a collection with the Collections API
- Creating a collection with the [Admin UI](#)
- [System Directories and Logs](#)

Using Collection Templates

This functionality is
not available with
LucidWorks Search
on AWS or Azure

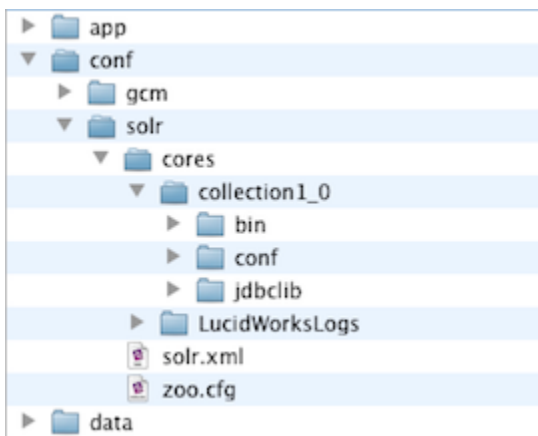
Collection templates allow you to copy the configuration files from a collection and use it as the basis for future templates. Creating a template is as simple as creating a `.zip` file from configuration files in the base collection and explicitly specifying that `.zip` file during new collection creation either via the [Admin UI](#) or the REST API.

Included Templates

Several templates are included with LucidWorks out of the box. They can be found in `$LWS_HOME/app/collection_templates`.

- `default.zip`: This has the same default options and out-of-the-box fields as the standard "collection1" that exists by default after LucidWorks Search installation.
- `essential.zip`: This is a stripped-down version of the LucidWorks default configuration that includes only the few fields that are absolutely essential for the system to run (see [Customizing the Field Schema](#) for more details on the default field set).
- `hadoop.zip`: provides the basic configuration for storing the Solr indexes for a collection in a Hadoop Filesystem (HDFS). For more details, see [Storing Indexes in HDFS](#).
- `lucidworkslogs`: provides the configuration for the LucidWorksLogs collection only. This is a system collection with a very specific configuration and this template should not be used for any other collection.

Creating a Template



To make a custom template, create a new collection and configure it as needed, whether that is via the user interface, using the REST API, or manual editing of configuration files. All of the configuration files for a collection reside in the `instance_dir` for the collection, which is found under `$LWS_HOME/conf/solr/cores/collection`, where `collection` is the name of the collection that is being used as the basis for the template.

Then create a .zip file from the `instance_dir`. The .zip file can have any name, including `default.zip`, although using the same name would overwrite the system default template, meaning it would not be available at a later time if needed. All templates must be placed in `$LWS_HOME/conf/collection_templates` to be available during collection creation.



We recommend that you use all the sub-directories from the `instance_dir` even if some of the files have not been customized in the base collection.

Related Topics

- [Working with Collections](#)

Indexing Documents

The first step to being able to search is to create an index. The index stores all the terms from documents in such a way that results for user queries can be returned as quickly as possible.

Indexes are created by breaking a document into individual words and saving the word list. At the same time, documents are not solely lists of sentences and words, but instead usually contain some sort of structure - an email will likely have "to" and "from" information; Word and PDF documents may have "title" and "author" information, in addition to the main "body"; product descriptions may have "price", "description" or "color" information. These are known as fields within each document. Adding field information to the word list facilitates a user's ability to search for emails from a specific person, or shoes that come in a particular color.

Fields can contain different types of data. A title field, for example, is usually text (character data). A price contains a mix of digits and special characters (such as \$ or €). Dates are generally. Defining the type of data that a field will contain is a critical first step in defining the fields for the index.

Defining Fields

There are several things to consider when configuring fields. The primary one is whether to store the field or not. Stored fields take up space in the index, but they allow the field to then be indexed (that is, made searchable) or available to users for display. It may be preferable to store a field and use it for display in a results list, but not allow it to be searchable. Alternately, a field can be designated for use in a facet, so it would be stored and indexed, but perhaps not searchable. A careful analysis of documents should occur before indexing to be able to anticipate how it will be indexed. If fields are not correctly configured before a document is indexed, documents will need to be re-indexed at a later time. If that is required, the existing index can be [deleted](#) and documents can be added to it from scratch.

Indexing Data Sources

In order for users to be able to search, LucidWorks Search needs to have indexed documents. LucidWorks Search supports two main approaches for document discovery:

- Documents can be pushed directly into the system. Users who are familiar with Solr may already have processes and systems in place to push documents into the index. This is also an option if LucidWorks Search is not able to connect to the repository to pull documents from it.
- Documents can be pulled from remote repositories. LucidWorks Search has several pre-defined types of repositories that it is able to connect to; you configure these connections by creating "data sources" and selecting options appropriate for your needs.

Each of these approaches has several options and caveats to consider, which are covered in more detail [Overview of Crawling](#).

Related Topics

- [Customizing the Field Schema](#)

How Documents Map To Fields

When LucidWorks Search crawls a data source, it extracts the target data and stores it in fields in the index. The specific mapping from the source data to the indexed fields is determined by the crawler you are using, which is in turn determined by the data source type. For a list of file types supported by LWE, see [Supported Filetypes](#). Let us consider two common file types, both processed by the [Aperture](#) crawler: a website and a Microsoft Word document.

For the [website](#), consider a case where you have crawled <http://www.lucidworks.com> with a crawl depth of zero, which means that only the first page is indexed. The Aperture crawler maps the web page as follows (note that this example is not complete or exhaustive):

Data Source	Field Mapping	Field Content	
url	url	http://lucidworks.com	
content-type	mimeType	html/text	
title	title	Lucid Imagination is now LucidWorks.	LucidWorks
body	body	The Future Of Search	

And so on.

For the [Microsoft Word document](#), consider this document, included here in its entirety:



Data Source	Field Mapping	Field Content
mimetype	mimeType	application/vnd.openxmlformats-officedocument.wordprocessingml
title	title	Example Word Doc

author	author	Drew Wheeler
body	body	This Is The Heading This is some text. It is very interesting.

For information on which crawlers handle which data source types, see the [Overview of Crawling](#). If using the [Admin UI](#), you don't need to worry about the crawler type. The UI also includes screens for modifying how documents are mapped to fields, or the Data Sources API can be used. For more information on fields in LucidWorks Search, see the Table of Fields in the section [Customizing the Field Schema](#).

Related Topics

- [Overview of Crawling](#)
- [Indexing Documents](#)
- [Editing Field Mapping](#)

Customizing the Field Schema

When indexing documents, LucidWorks Search doesn't merely generate a list of all the words found on the page. It also tries to recognize the structure of the document, and remember some of that structure in the index. The structure of indexed documents is represented by the fields defined for the LucidWorks Search index. When terms are saved in the index, they are saved with information about the field in which they were found in the document.

Field definitions are stored in a `schema.xml` file for each collection. Users familiar with Solr will recognize this file, since it is the same `schema.xml` file that is used with a Solr installation. Instead of editing this file by hand, however, LucidWorks Search allows modifying the field and field type definitions with the Admin UI or with the REST API.

By default, LucidWorks Search contains field definitions to support various features of LucidWorks (such as crawling documents and Click Scoring) and to make it easier for users to get up and running. Not all users will need all fields, however, so you may want to add fields unique to your search application or just to trim the default set of fields so the list is easier to work with. This section describes the default fields, how they are used by LucidWorks Search, and if they can be removed for local installations.

One of the primary added values of LucidWorks Search is the integration of content crawlers for web sites, filesystems and other repositories of content. Many of the default fields are for this purpose and should be retained. In many cases, if they are removed from the schema, they will be recreated the next time a crawler needs them. However, if not using the LucidWorks crawlers, they can generally be safely removed. They will be added based on a dynamic rule ("*" rule) in the `schema.xml` file that should be retained to avoid unexpected failures of the crawlers. If this rule is left in place, nearly any field in the schema can be removed as it will be added back if it is needed.



Only delete the "*" rule if you are absolutely positive other deleted fields will not be needed in your specific implementation. Deleting this rule may also complicate future upgrades, as it is not possible to predict when LucidWorks Search will add new fields to the `schema.xml` file to support future functionality.

- [Guidelines for Removing Fields from the Schema](#)
 - [Essential Fields](#)
 - [Built-In Search UI Fields](#)
 - [Fields to Support Specific Features](#)
 - [Crawler Fields](#)
 - [Other Dynamic Fields](#)
- [Table of Fields](#)

Guidelines for Removing Fields from the Schema

Essential Fields

There are two fields that must be retained in `schema.xml`. The Admin UI and the Fields API will not allow deleting them:

- `id`
- `timestamp`

There are three additional fields that are considered essential to LucidWorks Search.

- `data_source`
- `data_source_name`
- `data_source_type`
- `text_all`

The three data source-related fields are considered essential for the Admin UI and APIs to know the source of the content that has been indexed. If not using the Admin UI nor the LucidWorks REST APIs, they could be deleted.

The `text_all` field is required because `schema.xml` declares it as the default search field for the Lucene RequestHandler (query parser), which is also the default for the basic Solr query parser. If you are using `lucid` or `DisMax`, however, and will never use the Lucene or Solr query parsers, the field could be deleted. However, it may be best to retain it.



We have created a sample schema that includes only the essential fields listed above that can be used for collection creation. See [Using Collection Templates](#) for more information.

Built-In Search UI Fields

LucidWorks includes a default search UI that can be used as-is or replaced with a fully local interface. If using it as-is, even for testing or during initial implementation, the following fields must also be retained in `schema.xml`:

- `author`
- `author_display`
- `body`
- `dateCreated`
- `description`
- `keywords`
- `keywords_display`
- `lastModified`
- `mimeType`
- `pageCount`
- `title`
- `url`

The Search UI includes these fields for results display and default faceting, so for it to work properly, these fields should be retained.

Fields to Support Specific Features

Several fields are included in `schema.xml` in support of specific LucidWorks features. They can be removed if those features are disabled or not in use. In some cases, however, they will be added back to the schema if the feature is enabled in the future.

Feature	Fields
Click Scoring Relevance Framework	click click_terms click_val
ACL	acl
Spell Check	spell
Auto Completion	autocomplete
Enterprise Alerts	timestamp
SolrCloud and Near Realtime Search	_version_
De-duplication	signatureField

Crawler Fields

The crawlers included with LucidWorks create fields in `schema.xml` that begin with `*attr_*` and are used to store document-specific metadata during crawl processes. They are not generally used otherwise by LucidWorks (such as in search results or other computations). Due to the dynamic "*" rule, they will be added back to `schema.xml` if not in place. If not using the LucidWorks crawlers, they can be removed, but it is recommended to retain them if possible.

Other Dynamic Fields

Several other dynamic fields (all including an '*', such as `*_i`, `*_s`, `*_l`, etc.) are defined in `schema.xml`. These can be removed if they will not be used - the only two we recommend that you retain are the "*" rule and the `attr_*` fields.

Table of Fields



The table below notes whether a field will be indexed, stored, used for facets or included in results. This is default behavior, and can be modified locally. After customization, this table may not reflect the state of your `schema.xml` file.

Field Name	Type	Indexed	Stored	Used for Facets	Included in Results	Used for
------------	------	---------	--------	-----------------	---------------------	----------

version	long	X	X			Document version control, used with Near Realtime Search and SolrCloud .
acl	string	X	X			Storing Access Control List information.
attr_* (any field starting with 'attr_')	string	X	X			Created by the crawlers and used for a wide array of document-specific metadata. Not specifically declared in the schema.xml file, but dynamically created during crawls.
author	text_en	X	X		X	Raw author pulled from documents. Used by default in the built-in Search UI.
author_display	string	X		X		Used for display of authors in facets. Used by default in the built-in Search UI.

autocomplete	textSpell	X	X			Stores terms for the auto-complete index. By default, it is created by copying terms from the title, body, description and author fields.
batch_id	string	X	X			Identifies the batch that added the document.
bcc	text_en	X	X			Used in processing email messages.
belongsToContainer	text_en	X	X			Used to store the URL of the archive file (.zip, .mbox, etc.) which contains the file.
body	text_en	X	X			The body of a document (generally, the main text). Used by default for display in the built-in Search UI.

byteSize	int		X			The size of the document.
cc	text_en	X	X			Used in processing email messages.
characterSet	string		X			The character set used for the document. Only populated if it is declared in the document (most commonly with HTML files).
click	string	X	X			Used with the Click Scoring Relevance Framework and contains the boost value.

click_terms	text_ws	X	X			Used with the Click Scoring Relevance Framework and contains the top terms associated with the document.
click_val	string	X	X			Used with the Click Scoring Relevance Framework and contains a string representation for the boost value for the document. The format allows it to be used for processing function queries.
contentCreated	date	X	X			The creation date for the document, if available.
crawl_uri	string	X				A copy of the URL for the document.

creator	text_en	X	X			The creator of the document, if available.
data_source	string	X	X			The ID of the data source that crawled this document.
data_source_name	string	X	X	X		The name of the data source that crawled this document.
data_source_type	string	X	X		X	The type of data source that crawled this document.
dateCreated	date	X	X		X	The date the content was created, if available.
description	text_en	X	X		X	The description from a document, if it exists in the document. For example, Microsoft Office document properties contains a description field that can be filled in by the user.

email	text_en	X	X			Not currently used by any LucidWorks crawlers.
fileName	text_en	X	X			The name of the file.
fileSize	int	X	X			The size of the file.
from	text_en	X	X			Used in processing email messages.
fullname	text_en	X	X			Data in this field is mapped to "author".
generator	text_en	X	X			The name of the software that generated the document, if available.
id	string	X	X		X	Unique ID for the document.
id_highlight	text_en	X	X			No longer used by LucidWorks and will be removed in a later version.
incubationdate_dt	date	X	X			Used in older Solr example documents.
keywords	text_en	X	X		X	The keyword list from a Microsoft Office document.

keywords_display	comma-separated	X		X		Terms from the keyword field formatted for display to users.
lastModified	date	X	X		X	Date the content was last modified.
contentType	string	X	X	X	X	The type of document (PDF, Microsoft Office, etc.).
name	text_en	X	X			Data in this field is mapped to "title".
otherDates	date	X	X			Dates other than dateCreated or lastModified would be mapped to this field.
pageCount	int	X	X		X	The number of pages in a Microsoft Office document such as Word or PowerPoint.
partOf	string	X	X			Typically used for an email attachment, this points to the larger document of which this document is a part.
price	float	X	X			Example field that could be used for processing e-commerce data.

retrievalDate	date	X	X			Not currently used, but could be used for the date a web document was retrieved from its server.
rootElementOf	text_en	X	X			Populated only for the root or initial document of a crawl.
signatureField	string	X	X			Used with the de-duplication feature.
spell	textSpell	X				Stores the terms to be used in creating the spell check index. Created by copying terms from the title, body, description and author fields.
text_all	text_en	X				Used to combine text fields for faster searching. Created by copying terms from the id, url, title, description, keywords, author and body fields.
text_medium	text_en	X	X			Not currently used.
text_small	text_en	X	X			Not currently used.

timestamp	date	X	X	X	X	Time the document was crawled and used for date faceting and display of activities in the LucidWorks Admin UI. Also used for Enterprise Alerts to know when the document was added to the index for alerts processing.
title	text_en	X	X			The title of the document.
to	text_en	X	X			Used in processing email messages.
type	text_en	X	X			Used by the lucid.aperture crawler to store Aperture's classification of an information object, separate from its MIME type.
url	string		X		X	The URL to access the document.
username	text_en	X	X			No longer used and may be removed in a later version.

weight	float	X	X			Example field that could be used for processing e-commerce data.
--------	-------	---	---	--	--	--

Reindexing Content

It is considered a best practice to fully design your index (i.e., define all the fields you'll need and their attributes) before indexing large amounts of content. However, the reality is that things change - you have new requirements, new content, or you'd like to give users new options for searching.

As tolerant as LucidWorks Search is to changes, there are certain changes that cannot be made without fully reindexing, by which we mean deleting content from the indexes and re-processing it from scratch. Adding a field or changing field mapping options for an existing data source, as examples, require indexing the content again to get the new field information from the document or change the way the incoming content was processed into the index.

In addition, changes to the following attributes of a field require some degree of re-index:

- Field Type value
- If it is Indexed
- If it is Stored
- If it is Multi-valued
- Short Field Boost value

Below are the options for re-indexing content.

Re-crawl the Content

All of the crawlers store information about what documents it has previously processed, and uses that information for future crawls, usually only adding documents that are new (have never been indexed before), removed from the content repository (and should be removed from the index), or changed (and should be replaced in the index with the new copy). This means that documents already in the index are not re-processed and may be skipped, which may create a mis-match between existing content and new content being indexed.

Empty the Data Source

The [Admin UI](#) includes a button to Empty a data source. This button only deletes the documents from the data source, but does not reset any of the crawl history information, which keeps track of content that were previously found and uses that information to understand if content is new, has been deleted (and should be removed from the index), or has been updated (and should be removed and replaced with the new content). The associated API is the Collection Index Delete API, which has an option to specify deleting documents from the index associated with a data source.

If changes to a collection's field list or field type list have been made, emptying the documents from the data source may not be sufficient to fully re-crawl the content to update the fields because the next time a crawl is run it will be executed incrementally, using the crawl history information that it has stored. This means that if a document has not changed it will not be re-added to the index because the crawl history registers it as unchanged.

There is, however, a REST API to delete the crawl history called Data Source Crawl Data Delete which can be used if necessary.

Delete the Data Source

Deleting the data source deletes the metadata for the data source (the configuration details for LucidWorks Search to access the content repository), and any of the content from the index and the crawl history. It can be done with either the [Admin UI Delete button](#) or the Data Sources API. This might be the easiest way to clear the content so it can be re-crawled and re-indexed with the new field attributes.

Empty the Collection

Emptying the collection stops any running data sources, deletes the entire search index for the collection, and removes all crawl history for each data source. It is a good option if you have a number of data sources that you configured during initial implementation and would like to start fresh with production data. Emptying the collection can be done with either the [Empty this Collection button](#) in the Admin or the Collection Index Delete API.

Delete the Collection

Deleting the entire collection will delete all the data sources, stop any running jobs, delete all associated content, and remove all collection-related settings for the index. It can be done with the [Delete this Collection button](#) in the Admin UI or the Collections API.

Related Topics

- [Indexing Documents](#)
- [Overview of Crawling](#)

Multilingual Indexing and Search

LucidWorks Search has a number of capabilities designed to make working with multilingual data straightforward. By default, it includes support for most European languages, Japanese, Korean and Chinese. Multilingual capabilities are provided by Lucene's analysis process (see the [Language Analysis](#) section of the Solr Reference Guide for more details). Since Lucene is built on Java, which is Unicode enabled, many multilingual issues are handled automatically by LucidWorks and Solr. In fact, the main issues with multilingual search are mostly the same issues for working with any language: how to analyze content, configure fields, define search defaults, and so on.

Approaches to Multilingual Search

Besides the normal language issues, multilingual search does require decisions about whether to use a single field for each language, a field for each language or even a separate indexes for each language. Each of these three approaches has pros and cons.

Single Field Approach

Pros

- Simple to search across all languages
- Fast to search

Cons

- Requires Language Detection software, which is not included in LucidWorks, and which will slow down indexing
- Requires the query language to be specified beforehand, since language detection on queries is often inaccurate
- May return irrelevant results, since words may have same spelling but different meanings in different languages
- May skew relevancy statistics
- Hard to filter/search by language

Multiple Field Approach

Pros

- No language detection required
- Easy to search and/or filter by language
- Relevancy is clear since there is no noise from other languages with common spellings (minor)

Cons

- Many languages = many fields = more difficult to know what to search
- Slower to search across all languages

Multiple Indexes Approach

Pros

- Easy to bring one language off-line for maintenance without effecting other languages
- Can easily partition data and searches across machines by language
- Easy to search and filter by language

Cons

- More complex administration
- Slower and more difficult to search across all languages

Currently, LucidWorks supports the multiple field and multiple index approach out of the box, but the single field approach is still possible with some additional work that requires intermediate level Solr expertise.

Open Source Multilingual Capabilities

The crux of multilingual handling is applying analysis techniques to the content to be indexed. These techniques are specified in the Solr's `schema.xml` by the `<fieldType>` declarations. Out of the box, LucidWorks comes configured with numerous predefined field types designed to make indexing and searching multilingual content easy to do.

Note that most of the supported languages (especially the European languages) are designed to use Dr. Martin Porter's [Snowball stemmers](#) along with stop word filters, synonym filters and various other filters.



Multiple Languages May Require Customization

Although LucidWorks ships with default analysis and filter techniques, they may need customization for your search application. Consider the included language configurations to be good starting points for support of any given language and make adjustments as needed. For information on relevance tuning and debugging for additional tools and techniques to improve results, see [Understanding and Improving Relevance](#).

By setting up the appropriate fields per language, it is possible to simply point LucidWorks at the given data source and have it index the content.

Adding Support for Other Languages

While there are a wide variety of languages available "out of the box", there may come a time where support for a new language is needed. There are a few possibilities:

- Try out the language with the StandardAnalyzer, since it often does the right thing as far as tokenization and basic analysis goes. Note that the analyzer doesn't do stemming or perform more advanced language translation.

- Write an Analyzer, Tokenizer or TokenFilter and the associated Solr classes as described on the Solr Wiki page at <http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>.
- Use an n-gram character-based approach that chunks characters into n-grams and indexes them. Accuracy will be limited, but it may be better than nothing.

If choosing the second option, the new capability can be brought into LucidWorks as described in the Solr wiki section on [SolrPlugins](#).

Related Topics

- [Language Analysis](#) from the Solr Reference Guide
- [AnalyzersTokenizersTokenFilters](#) from the Apache Solr Wiki

Lucid Plural Stemming Rules

The purpose of stemming is to translate different forms of similar words to a common form so that a query for one form of a word will also match the other forms. The most common difference between word forms is singular words versus their plurals. Another variation in form is the variety of conjugations of a word. Although the administrator can select what stemming filter or options are enabled for each field type, by default all text fields will have a stemming filter that converts most plural words to singular.

Stemming is not a perfect process, so some plurals may be missed and some singular words may be mistakenly translated to some other singular or possibly even a non-word. Non-words, such as jargon, names, and acronyms can also be mistakenly stemmed. But, since stemming usually occurs at both document indexing time and at query time, improper stemming is frequently not even detectable. The default rules try to avoid removing "s" endings that are not plural (or verb conjugations), such as "alias" or "business."

If stemming proves problematic for a given application, the administrator can always turn it off or select an alternative stemming filter.

The Lucid plural stemmer is designed to focus on stemming of plural words into their singular forms. It is rule-based, so the rules can be supplemented and tuned to handle a wide range of exceptions. Individual words can be protected from stemming and can be given special-case stem words. Usually, general patterns cover wide classes of words.

The input token does not need to be lower case, but the stemming change will be lower case.

The Stemming Rules File

The default rules file is named `LucidStemRules_en.txt` and found in `$LWS_HOME/conf/solr/cores/collection/conf`. The rules file can be defined by changing the "rules" parameter in `schema.xml` for `com.lucid.analysis.LucidPluralStemFilterFactory`. These rules files are specified per text field type. It is expected that each natural language will have its own stemming rules file. This file is also specific to each collection.

If you wish to edit the stemming rules file, adhere to the following format guidelines.

- An exclamation point (!) indicates a comment or comment line to be ignored.
- White space is extraneous and ignored.
- Blank lines ignored.

Rules are evaluated in the order that they appear in the rules file, except that whole protected words and replacement words are processed before examining suffixes.

To restrict the minimum word length that is to be stemmed, simply create rules consisting of only question marks ('?') to match and protect words of those lengths. For example, to protect words of less than four characters in length, add three rules, before any other rules:

<pre>? ! Protects 1-char words. ?? ! Protects 2-char words. ??? ! Protects 3-char words.</pre>
--

Types of Stemming Rules

Protected Word

Just write the word itself, it will not be changed.

- word

Replacement Word

Word will always be changed to a replacement word.

- word => new-word
- word -> new-word
- word --> new-word
- word = new-word

Protected Suffixes

Any matching word will be protected.

- pattern suffix

Pattern may start with an asterisk to indicate variable length. Use zero or more question marks to indicate that a character is required. Use a trailing slash if a consonant is required.

Examples:

- ?ass
- *??ass
- *???/ass

Translation Suffix

The suffix of a matching word will be replaced with new suffix.

- pattern suffix => new-suffix

Pattern rules are the same as for protected suffixes. The pattern may be repeated before the replacement suffix for readability.

Examples:

- *ses => se
- *ses -> *se

- */uses => se
- *???s =>
- *???s => *

The latter two examples show no new suffix, meaning that the existing suffix is simply removed.

Example Stemming Rules File

Here is the default `LucidStemRules_en.txt` file that ships with LucidWorks Search, found in `$LWS_HOME/conf/solr/cores/collection/conf` (unique to each collection):

```
? ! Minimum of four characters before any stemming.
??
???
*ss ! No change : business
*'s ! No change : cat's - Handled in other filters.
*elves => *elf ! selves => self, elves, themselves, shelves
appendices => appendix
*indices => *index ! indices => index, subindices - NOT jaundices
*theses => *thesis ! hypotheses => hypothesis, parentheses, theses
*aderies => aderie ! camaraderie
*ies => *y ! countries => country, flies, fries, ponies, phonies, queries, symphonies
*hes => *h ! dishes => dish, ashes, smashes, matches, batches
*???oes => *o : potatoes => potato, avocados, tomatoes, zeroes
goes => go
does => do
?oes => *oe ! toes => toe, foes, hoes, joes, moes - NOT does, goes - but "does" is also
plural for "doe"
??oes => ??oe ! floes => floe
*sses => *ss ! passes => pass, bosses, classes, presses, tosses
*igases => *igase ! ligases => ligase
*gases => *gas ! outgases => outgas, gases, degases
*mases => *mas ! Christmases => Christmas, Thomases
?vases => *vas ! canvases => canvas - NOT vases
*iasess => *ias ! aliases => alias, bias, Eliases
*abuses => *abuse ! disabuses => disabuse, abuses
*cuses => *cuse ! accuses => accuse, recuses, excuses
*fuses => *fuse ! diffuses => diffuse, fuses, refuses
*/uses => *us : buses => bus, airbuses, viruses; NOT houses, mouses, causes
*xes => *x ! indexes => index, axes, taxes
*zes => *z ! buzzes => buzz
*es => *e ! spaces => space, files, planes, bases, cases, races, paces
*ras => *ra ! zebras => zebra, agoras, algebras
*us
*/s => * ! cats => cat (require consonant (not "s") or "o" before "s")
*oci => *ocus ! foci => focus
*cti => *ctus ! cacti => cactus
plusses => plus
gasses => gas
```

```
classes => class
mice => mouse
data => datum
!bases => basis
amebiases => amebiasis
atlases => atlas
Eliases => Elias
molasses
feet => foot
backhoes => backhoe
calories => calorie

! Some plurals that don't make sense as singular
sales
news
jeans
```

Choosing an Alternate Stemmer

Out of the box, the Lucid query parser comes with a basic plural stemmer that translates most plural words to their singular form. This should be sufficient for most applications. The stemming rules are all rule-based in an easy to read and write text file format that permits the addition of new rules and permits words to be protected or mapped specially. This permits flexibility for many more specialized applications.

If for some reason the administrator wishes to use an alternative stemmer, the change can be made manually in the `schema.xml` file or by using the FieldTypes API. Any stemming filter can be specified, but Lucid KStem is a typical alternative.



Information for LucidWorks Search in the Cloud Users

The instructions below refer to editing `schema.xml` to modify the stemmer used for each field type. Manual editing of the `schema.xml` file cannot be done by customers using LucidWorks Search hosted on AWS or Azure, but the same results can be achieved with the FieldTypes API.

Be sure to use the same stemmer class for both the index and query analyzers. If the stemmer classes do not match, the result can be that some queries can fail if terms were indexed according to different rules than those used by the Lucid query parser.

In general, it is best to *delete* the index and do a full re-indexing of the data collection whenever an *index* analyzer is radically changed, such as is the case when stemming filters or rules are changed. See [Reindexing Content](#) for more information about the options to reindex.

Other alternative stemming filters, such as Snowball and Porter, can be used instead of Lucid KStem if desired.

Using the FieldTypes API

The FieldTypes API is covered in depth in the section on the FieldTypes API.

The stemming rules are defined in the "analyzers" section for the field type. The analyzers section is considered an individual attribute as a whole, and it's not possible to update a single part of the analyzers rules without updating the entire section.

The `com.lucid.analysis.LucidPluralStemFilterFactory` class represents the default plural stemmer and will be shown in an API call in both the `index` and `query` section of the `analyzers` attribute. The `rules` parameter specifies the name of the text file that contains the plural stemming rules.

The `com.lucid.analysis.LucidKStemFilterFactory` class represents the Lucid KStem stemmer. To switch to this stemmer (or any other), make an API PUT call to the appropriate field type and update the `analyzers` attribute (in both the `index` and `query` sections).

For example, changing to the Lucid KStem stemmer for the `text_en` field type would require the following API call:

```
curl -X PUT -H 'Content-type: application/json'
-d '{"analyzers": {
  "index": {
    "char_filters": [ ],
    "token_filters": [
      {
        "catenateAll": "0",
        "catenateNumbers": "1",
        "catenateWords": "1",
        "class": "solr.WordDelimiterFilterFactory",
        "generateNumberParts": "1",
        "generateWordParts": "1",
        "splitOnCaseChange": "1"
      },
      {
        "class": "solr.LowerCaseFilterFactory"
      },
      {
        "class": "solr.ASCIIFoldingFilterFactory"
      },
      {
        "class": "com.lucid.analysis.LucidKStemFilterFactory"
      }
    ],
    "tokenizer": {
      "class": "solr.WhitespaceTokenizerFactory"
    }
  },
  "query": {
    "char_filters": [ ],
    "token_filters": [
      {
        "class": "solr.SynonymFilterFactory",
        "expand": "true",
        "ignoreCase": "true",
        "synonyms": "synonyms.txt"
      },
      {
        "class": "solr.StopFilterFactory",
        "ignoreCase": "true",
        "words": "stopwords.txt"
      },
      {
        "catenateAll": "0",
        "catenateNumbers": "0",
        "catenateWords": "0",
```

```

        "class": "solr.WordDelimiterFilterFactory",
        "generateNumberParts": "1",
        "generateWordParts": "1",
        "splitOnCaseChange": "1"
    },
    {
        "class": "solr.LowerCaseFilterFactory"
    },
    {
        "class": "solr.ASCIIFoldingFilterFactory"
    },
    {
        "class": "com.lucid.analysis.LucidKStemFilterFactory"
    }
],
"tokenizer": {
    "class": "solr.WhitespaceTokenizerFactory"
}
}
}}' http://localhost:8888/api/collections/TestCollection/fieldtypes/text_en

```

Editing schema.xml

If you edit `schema.xml`, and search for the `text_en` field type, you should see that both its index and query analyzers have XML entries for the stemming filter that appear as follows:

```

<filter class="solr.ISOLatin1AccentFilterFactory"/>
<!-- <filter class="com.lucid.analysis.LucidKStemFilterFactory"/> -->
<filter class="com.lucid.analysis.LucidPluralStemFilterFactory"
rules="LucidStemRules_en.txt"/>

```

The `com.lucid.analysis.LucidPluralStemFilterFactory` class represents the default plural stemmer. The `rules` parameter specifies the name of the text file that contains the plural stemming rules.

The `com.lucid.analysis.LucidKStemFilterFactory` class represents the Lucid KStem stemmer, which is disabled by default using the standard `<!-- and -->` comment markers.

To disable the default plural stemmer and enable Lucid KStem, simply remove the comment markers from the latter and add them to the former. Do this same thing for both the index and query analyzers. The edited lines should now appear as follows:

```

<filter class="solr.ISOLatin1AccentFilterFactory"/>
<filter class="com.lucid.analysis.LucidKStemFilterFactory"/>
<!-- <filter class="com.lucid.analysis.LucidPluralStemFilterFactory"
rules="LucidStemRules_en.txt"/> -->

```

Deleting the Index

During application development, you might use sample data that is inappropriate for the production system. To remove this data, you can delete the entire index or just delete the content and crawl history for a single data source.

The easiest way to do this is to use the [Admin UI](#) to delete documents from a specific data source or an entire collection.

Another way to do this is to issue an API command using the Collections Index API. This API provides two methods to stop all running indexing tasks, clear the index, and clear any persistent crawl data (crawl history) for either the entire collection or a single data source.



This Will Delete ALL of Your Data

The following procedure to delete a collection should only be used if you are sure you want to delete **all documents** in your index. Once this command has been executed, there is **no way** to retrieve the content. If only some documents should be deleted, use the method to delete documents for a specific data source.

If you only want to clear the crawl history, the Data Source Crawl Data API provides a way to delete only the history for a data source, but not the content.

An alternative approach would be to issue a delete command directly to Solr with the following syntax. However, this will not stop running tasks nor clear persistent crawl data.

```
http://localhost:8888/solr/update?stream.body=<delete><query>id:[\* TO  
\\*\]</query></delete>
```

Related Topics

- [Reindexing Content](#)
- [Overview of Crawling](#)

Storing Indexes in HDFS

It is possible to store the Solr indexes in your Hadoop Filesystem (HDFS). The benefits of this are to distribute the indexes and Solr's transaction logs across a Hadoop cluster. Note that this does not use MapReduce for index processing, but instead uses Hadoop for transaction log and index file storage. LucidWorks Search (and Solr) support doing this with Hadoop version 2.x only.

In LucidWorks Search, this is enabled by with a new [collection template](#) named "hadoop" which defines the configuration required to store Solr indexes on Hadoop. This template can be used to create new collections whose indexes will be stored in the HDFS specified with the parameters.

The main configuration changes are defined in `solrconfig.xml`. The `directoryFactory` needs to be set to use the `HdfsDirectoryFactory` and two parameters are defined for `solr.hdfs.home` and `solr.hdfs.confdir`. The `solrconfig.xml` supplied with the 'hadoop' collection template includes this section:

```
<directoryFactory name="DirectoryFactory"
class="org.apache.solr.core.HdfsDirectoryFactory">
  <str name="solr.hdfs.home">${solr.hdfs.home}</str>
  <str name="solr.hdfs.confdir">${solr.hdfs.confdir}</str>
</directoryFactory>
```

Note that the two required parameters are defined as system properties. To supply the values for the system properties, you should modify `$LWS_HOME/conf/master.conf` for the installation to add them. The values must be supplied for the [LWE-Core component](#) as in this example:

```
# JVM Settings for LWE-Core
lwecore.jvm.params=-Xms512M -Xmx1024M -XX:MaxPermSize=256M -Duser.language=en
-Duser.country=US -Duser.timezone=UTC -Dfile.encoding=UTF-8
-Dcom.sun.management.jmxremote -Dsolr.hdfs.home=/path/to/hadoop/home
-Dsolr.hdfs.confdir=/path/to/hadoop/home/conf
```

Defining the values in `master.conf` has the benefit of allowing you to define the HDFS location once. However, if you have multiple HDFS locations, you could instead define the values within the `solrconfig.xml` file for each collection that will be stored in HDFS. In that case, do not also add the values to `master.conf`.

Note the parameters described here are the basic parameters to allow LucidWorks to store the Solr indexes on HDFS. There are other available parameters, however, described in the Apache Solr Reference Guide section [Running Solr on HDFS](#).

Related Topics

- [Using Collection Templates](#)
- [Running Solr on HDFS](#) from the Apache Solr Reference Guide

Crawling Content

This section describes how to configure crawling with LucidWorks Search, to get the content to put in the indexes.

For the most part, crawling only requires configuring a data source with the UI or the API and starting the crawl. However, if using batch crawling, Access Control Lists, databases containing binary data, or an "external" crawler, there may be additional configuration you'll want to do.

Start with the [Overview of Crawling](#) to understand how the crawlers work.

Then dive into the detailed sections as needed:

- [Supported Filetypes](#)
- [Troubleshooting Document Crawling](#)
- [Crawling Windows Shares with Access Control Lists](#)
- [Indexing Binary Data Stored in a Database](#)
- [Using the High Volume Crawlers](#)
- [Suggestions for External Data Sources](#)
- [Indexing Documents Directly to Solr](#)
- [Integrating Nutch](#)
- [Processing Documents in Batches](#)

Overview of Crawling

LucidWorks Search has integrated several crawlers to make adding content to the index easier and more straightforward.

A *crawler* is a program which understands how to connect to a remote repository (or several types of repositories), find documents within the repository, and retrieve the documents for indexing by the system. A synonym in some contexts is a *connector*, but there are differences between the terms. A crawler discovers new documents on its own and makes decisions about which documents to retrieve, based on rules provided to it by its own code or by configuration. A connector is more passive - it connects to a repository and pulls all the documents, without the ability to make decisions; interpreting rules and making decisions would be up to the crawler which controls the connector.

As each repository is different, each crawler needs information to connect to a specific repository, such as the network address of the repository and any required authentication information. This information is provided to the crawler by creating a *data source*.

The data source is the central way in which you interact with the crawlers. There is one defined per repository, filesystem, website, etc. So, for example, if you want to index three websites, you'll create three Web Data Sources. Three S3 buckets, then you'll create three S3 Data Sources.

For the most part, we've tried to make each data source consistent in terms of the options provided, but there are differences between the crawlers and their capabilities. This leads to differences when configuring data sources of different types, and differences in performance and behavior of the crawlers themselves while retrieving documents and passing them along the indexing process.


The Crawl Process

When starting a crawl, the crawler associated with a specific data source uses the information saved in the data source configuration to connect to the repository and find documents. Most of the data sources support inclusion or exclusion parameters to define the types of documents (or paths to documents) that should be indexed. The crawlers use that information to know what pages to retrieve for eventual indexing.

Topics covered in this section:

- [The Crawl Process](#)
 - [Re-Crawling Documents](#)
- [Data Source Options](#)
 - [Logging](#)
 - [Scheduling](#)
 - [Field Mapping](#)
- [Data Source Types](#)
- [Related Topics](#)

The crawlers do not actually index content. A crawler retrieves the pages, and passes them to a *parser*, which prepares the documents for the indexing process. The parser handles breaking the documents into their parts, identifying fields within the documents and normalizing data so it can be more easily consumed in the index. In most cases, the crawlers use Apache Tika for parsing.

 The exception to this is the Aperture crawler, which has its own parser embedded within it. In cases where the Aperture parser fails to parse a document, Tika is used as a fall-back. However, documents that were successfully parsed by the Aperture crawler do not get another pass through Tika. There is no way to change this behavior at this time.

Once documents have been retrieved and parsed, they are passed to the *UpdateController* which pushes them into the index using SolrJ, a common client used for indexing content in Solr. This process also performs *field mapping*, where the extracted fields from a document can be mapped to other fields.

Re-Crawling Documents

When working with data sources and their content, it helps to understand how content is handled during the initial crawl and in subsequent re-crawls to update the index with new, updated, or removed content.

Some of the crawlers keep track of documents that have been "seen" which helps speed later crawls by not processing unchanged content, but it can be confusing if the configuration settings change between crawls. In some cases, you may need to remove the crawl history in order to get the results you want; an example of this would be the `add_failed_docs` setting: if it is not set for the initial crawl of a repository, it will be skipped on subsequent crawls unless it has been modified in some way. Other examples include (but aren't limited to) settings to map fields from the incoming documents to another field, options to add LucidWorks-specific fields to the documents, as well as changes to fields themselves and any dynamic field rules.

If making changes to a data source configuration after content has already been crawled and indexed, review the options in the section on [Reindexing Content](#) for possible approaches.

[Back to Top](#)

Data Source Options

Logging

The crawlers log information about attempts to access documents and the results of those attempts. The log is kept in `$LWS_HOME/data/logs` in a file named `connectors.<YYYY_MM_DD>.log`.

In general, the crawlers will:

- print one line to the log with the document ID when it has successfully accessed a document, describing the status (New, Updated, Deleted, etc.). In cases where the document could not even be accessed, this may lead to the attempt not being recorded in the logs.
- not log documents of unknown type that cannot be processed as plain text.
- not log documents that fail parsing.
- not add documents that fail parsing.

Each of these behaviors can be changed in most crawlers, which would allow more information to be added to the log or more documents added to the index. With some crawlers, however, the default behaviors are the only options. More information for each data source type is available in the documentation for the [Admin UI](#) and the REST API.

Scheduling

Each data source can be scheduled to run at regular intervals. Using the Admin UI, it is only possible to schedule crawling at specific intervals (hourly, daily, weekly), but using the REST API, more complex schedules can be constructed. It is, however, only possible to have a single schedule for each data source.

Field Mapping

Field Mapping provides the ability to map fields in documents to fields or dynamic field rules already defined in LucidWorks or add fields to incoming documents. This can be done generically when an unexpected field is introduced or specifically for known incoming fields. The mapping rules can be manipulated via the Admin UI from the [Data Source Details](#) screen, or with either the Data Sources API or the Field Mapping API.

Some explicit field mappings are defined by default. This table shows the LucidWorks Search default mappings:

From Crawler Metadata	To Field
acl	acl
author	author
batch_id	batch_id
body	body
content-encoding	characterSet
content-length	fileSize
contentcreated	dateCreated
contentlastmodified	lastModified
contributor	author
crawl_uri	crawl_uri

created	dateCreated
creator	creator
date	null
description	description
filelastmodified	lastModified
filename	fileName
filesize	fileSize
fullname	author
fulltext	body
keyword	keywords
last-modified	lastModified
last-printed	null
lastmodified	lastModified
lastmodifiedby	author
links	null
messagesubject	title
mimetype	mimeType
name	title
page-count	pageCount
pagecount	pageCount
plaintextcontent	body
plaintextmessagecontent	body
slide-count	pageCount
slides	pageCount
subject	subject
title	title
type	null
url	url

When the mapping is created or updated, LucidWorks checks the mappings against the `schema.xml` for the collection and verifies that the target fields exist in the schema.

During indexing, the field mapping process performs the following steps:

1. The mappings are checked for the existence of the source field name. If it exists, it will be mapped to the target field.
2. If the source field name does not exist in the mappings, the `schema.xml` for the collection is checked. If the source field name exists in the schema, it will be indexed to that field.
3. If a `dynamic_field` has been defined, a dynamic field will be created according to the dynamic field rule.
4. If a `default_field` has been defined, the source field will be mapped to the defined default field.
5. If none of these steps has produced a match, the field will be discarded.

[Back to Top](#)

Data Source Types

LucidWorks Search currently supports 8 crawlers and 13 types of data sources. When using the Admin UI, the selection of a crawler is hidden; when using the REST API, the selection of a crawler is a required attribute.

The table summarizes the types of content repositories that can be crawled:

Crawler	Data Source Types Supported	Capabilities	Limitations (not comprehensive; see documentation for each type for full details)
---------	-----------------------------	--------------	---

Aperture	<ul style="list-style-type: none">• Websites• Filesystems	<ul style="list-style-type: none">• Can crawl websites and filesystems.• Stores a history of documents that have been seen before.• Indexes data contained in <META> tags.• Web crawling will respect robots.txt rules or can be configured to ignore them.	<ul style="list-style-type: none">• The Aperture crawler is not designed for large-scale crawls of more than about 10,000 pages or files in a single crawl.• It is a single-threaded process, meaning that one data source will only use a single server process to crawl sites. This can make a long crawl take a long period of time to complete.• Multiple data sources all use the same "triple-store", which is a database inside Aperture that keeps track of web pages visited. If multiple data sources are running at the same time, the triple-store can get easily corrupted. It's highly recommended to avoid running multiple Aperture-based crawls at the same time.• Doesn't use Apache Tika for document parsing and may not be as accurate with some documents as Tika (however, if it cannot parse a document at all, it will pass that document to Tika for parsing).
----------	--	--	---

JDBC	<ul style="list-style-type: none">• Databases	<ul style="list-style-type: none">• Allows indexing of databases.• Supports nested queries for complex data environments.• Supports delta queries to limit subsequent crawls on only new or changed table rows.	<ul style="list-style-type: none">• The LucidWorks Search implementation is based on the DataImportHandler, which can be difficult to precisely configure in unique environments.• Requires uploading a driver before it can be used.• Converting date types can be problematic.
Google Connector Manager	<ul style="list-style-type: none">• SharePoint Repositories	<ul style="list-style-type: none">• Indexes all content in the SharePoint repository (files, discussion boards, calendars, contacts, sites, images, etc.).• Support SharePoint security configuration.• Can add new connectors supported by the Google Connector Manager framework.	<ul style="list-style-type: none">• Must install additional Web services to work properly.• Security options can be complex to configure.

SolrXML	<ul style="list-style-type: none">• SolrXML files	<ul style="list-style-type: none">• Easy to understand XML structure.• Many users already have documents in this format due to prior use of Solr.• Can point it to a directory of files instead of one at a time.• Can add a unique identifier to each document as it's indexed if it doesn't have one already.	<ul style="list-style-type: none">• Not a generic XML indexer; documents must be structured in a very specific way.
Filesystem	<ul style="list-style-type: none">• Amazon S3 buckets• SMB/Windows Shares• Hadoop Distributed Filesystems (HDFS)• Hadoop over S3• FTP servers• Local Filesystems	<ul style="list-style-type: none">• Provides access to multiple remote filesystems.• Allows multi-threaded crawls.	<ul style="list-style-type: none">• Must allow the LucidWorks server access to the remote systems.• Hadoop crawls are throttled to prevent overloading the system.• Crawls are "stateless", meaning the crawler can not be aware of documents deleted or modified between crawls. In crawl statistics, this can mean that deleted or updated documents are not counted as such, or that adding the total number of "new" documents in two different crawls does not equal the number of documents in the index.

MongoDB	<ul style="list-style-type: none">• MongoDB	<ul style="list-style-type: none">• Supports multiple databases and tables within a single MongoDB installation	<ul style="list-style-type: none">• MongoDB collections indexed restricted by username and password provided to the crawler.• Crawling all databases and collections requires allowing the crawler to have "admin" access to the database.
MapR	<ul style="list-style-type: none">• MapR filesystems	<ul style="list-style-type: none">• Supports all features of the HDFS crawler, but is optimized for MapR Hadoop distributions.• Allows multi-threaded crawls.	<ul style="list-style-type: none">• Only crawls MapR distributions; other Hadoop distributions must use the generic HDFS crawler.
MapR High Volume	<ul style="list-style-type: none">• MapR Filesystems	<ul style="list-style-type: none">• Allows unthrottled crawling of MapR filesystems.• Supports adding extracted document metadata and Behemoth annotations, if present.• Customized client for MapR specifically.	<ul style="list-style-type: none">• Must design your LucidWorks Search cluster appropriately to take full advantage of the speed capabilities.• The LucidWorks Search implementation is still in beta phase; may include unknown bugs.• Only crawls MapR distributions; other Hadoop distributions must use the generic High Volume HDFS crawler.

Azure Blob	<ul style="list-style-type: none">• Azure Blob storage	<ul style="list-style-type: none">• Indexes all content found in an Azure Blob storage container.	<ul style="list-style-type: none">• Can only specify a single container.
Azure Table	<ul style="list-style-type: none">• Azure Table instances	<ul style="list-style-type: none">• Indexes all content found in an Azure Table instance.	<ul style="list-style-type: none">• Does not support incremental crawling (i.e., delta queries). All documents are retrieved with every crawl.
Twitter Stream	<ul style="list-style-type: none">• Twitter Stream API	<ul style="list-style-type: none">• Allows filtering indexed tweets by userID, location, or keywords.	<ul style="list-style-type: none">• Will continue to crawl indefinitely unless manually stopped or controlled with a parameter that's only available via the REST API.• The LucidWorks implementation is still in beta phase; may include unknown bugs.
High-Volume HDFS	<ul style="list-style-type: none">• Hadoop Distributed Filesystems (HDFS)	<ul style="list-style-type: none">• Allows unthrottled crawling of HDFS systems.• Supports adding extracted document metadata and Behemoth annotations, if present.	<ul style="list-style-type: none">• Must design your LucidWorks cluster appropriately to take full advantage of the speed capabilities.• The LucidWorks implementation is still in beta phase; may include unknown bugs.

External	<ul style="list-style-type: none">• Push to LucidWorks	<ul style="list-style-type: none">• Can use SolrJ or another established Solr indexing process to get documents into LucidWorks.• Full access to field mapping capabilities that other crawlers use.	<ul style="list-style-type: none">• The documents or processes for crawling must be prepared in advance.
----------	--	---	--

Related Topics

- [Data Sources in the Admin UI](#)
- Data Sources with the REST API
- Custom Connector Guide

[Back to Top](#)

Supported Filetypes

LucidWorks Search crawlers can identify many different file formats (MIME types), and can extract text and metadata from the MIME types listed in the table below. Even if the crawlers cannot extract data from a file, it can often at least recognize the file type and index basic information about the file, such as the filename and its metadata. Many of the crawlers have settings that allow how to handle the situation where the MIME type is not supported.

Note that extracting data from third party proprietary file formats is often difficult and may result in irregular text being extracted and indexed. If you encounter a format that is supported, but does not get properly extracted, please share the information with Lucid Support, including the file, if possible.

Supported File Formats

Name	MIME Type(s)	Notes
HTML	text/html	
Images	image/jpeg, image/png, image/tiff	Metadata Only
Mail	message/rfc822 and message/news	Some mime based mail attachments can be extracted.
MP3 Metadata	audio/mpeg	Metadata only
Microsoft Office	Word, PowerPoint, Excel, MS Publisher, Visio	All applications are trademarks of the Microsoft Corporation
Open Office	OpenDocument and StarOffice documents	
OpenXML	Microsoft's latest Office format	
Adobe Portable Document Format	application/pdf	PDF is a trademark of Adobe
Plain Text	text/plain	
Quattro	application/x-quattropro, application/wb2	Trademark of Corel
Rich Text Format	text/rtf	
eXtensible Markup Language (XML)	text/xml	
Archives	application/zip, application/gzip, application/x-tar	

Troubleshooting Document Crawling

LucidWorks Search crawling events are logged to the `connectors.<YYYY_MM_DD>.log` file, found in the `$LWS_HOME/data/logs` directory.

Serious exceptions will be reported to the LucidWorksLogs collection, which you can search as you can any other collection through the default Search UI. In addition, the Admin UI provides some visibility into errors during crawling by showing them on the **Server Log** page, found under the Status menu. That page also allows access to browse all the log files without having to access the server.

Problems such as a document not being found or access denied will not be reported the the LucidWorksLogs collection, but will show in the Admin UI and in the Data Source Status/History APIs as "not found". This may make it difficult to find which documents were skipped, but a review of the log file may yield further information.

In general, the crawlers will:

- print one line to the log with the document ID when it has successfully accessed a document, describing the status (New, Updated, Deleted, etc.). In cases where the document could not even be accessed, this may lead to the attempt not being recorded in the logs. This can be changed by modifying the setting "Log Extra Detail" in crawlers that support it.
- not log documents of unknown type that cannot be processed as plain text. This can be changed by modifying the setting "Log warnings for unknown mime types" in crawlers that support it.
- not log documents that fail parsing. This can be changed by modifying the setting "Fail unsupported file types" in crawlers that support it.
- not add documents that fail parsing. This can be changed by modifying the setting "Add failed docs" in crawlers that support it.

- By default, the LucidWorks Search Connectors log does not record the collection or data source associated with crawl activities. However, if you would like to record that information to make troubleshooting simpler, you can edit the `$LWS_HOME/conf/log4j-connectors.xml` file.

In the file, find the section that begins with a comment to "Use the pattern below to log additional context info...", as below:

```
<!-- Use the pattern below to log additional context info like collection and
data source name -->
<!--
    <param value="%d{ISO8601} %p %c{2} - %X %m%n" name="ConversionPattern"/>
-->
```

Uncomment `<param value="%d{ISO8601} %p %c{2} - %X %m%n" name="ConversionPattern"/>` and save the file. You should [restart](#) LucidWorks Search after making this change.

Errors Creating Data Sources

Path or URL Errors

By default, all data sources try to verify that the repository to be crawled is accessible to the Connectors component with the information provided. In most cases, the data source will not be created unless the data source is accessible.

Most of the crawlers support disabling the verification step during data source creation with a parameter in the API (the Admin UI has no ability to skip verification). However, if the Connectors component cannot access the repository, it will not be able to crawl it.

MapR-related Errors

Before using either MapR data source, you must first have the MapR client installed at a filesystem location accessible by the LucidWorks Connector component. For information about the MapR client, please see the MapR documentation [Setting Up the Client](#).

The Connector component looks for the client libraries in `/opt/mapr` by default, but the location can be modified by editing the `lweconnectors.jvm.params` in `$LWS_HOME/conf/master.conf`. Find the setting `-Dmapr.home` and modify the path as needed.

The following errors indicate that either the MapR Client is not installed or not accessible to the Connectors component:

In `core.<date>.log`:

- Unprocessable Entity (422) - [{"message":"unknown crawler type lucid.map.reduce.maprfs","code":"error.invalid.value","key":"crawler"}]
- Unprocessable Entity (422) - [{"message":"unknown crawler type lucid.mapr","code":"error.invalid.value","key":"crawler"}]

In connectors.<date>.log:

- External library path doesn't exist: /opt/mapr/hadoop/hadoop-0.20.2/conf
- External library path doesn't exist: /opt/mapr/hadoop/hadoop-0.20.2/lib
- External library path doesn't exist:
/opt/mapr/hadoop/hadoop-0.20.2/lib/jsp-2.1
- No valid external paths - skipping mapr-client initialization.
- Dependency 'mapr-client' of /LucidWorks/2.5.6-32/app/crawlers/mapr-crawler.jar NOT FOUND
- No valid crawler plugins in
file:/LucidWorks/2.5.6-32/app/crawlers/mapr-crawler.jar
- Dependency 'mapr-client' of
/LucidWorks/2.5.6-32/app/crawlers/mapr-hv-crawler.jar NOT FOUND
- No valid crawler plugins in
file:/LucidWorks/2.5.6-32/app/crawlers/mapr-hv-crawler.jar

Exact paths referenced in these errors will vary depending on how you have installed LucidWorks Search.

Understanding Crawl Errors

Crawling is dependent on a number of factors. In order for a site to be crawl-able, several things must be aligned:

- The repository must be supported by one of the crawler and data source types.
- The repository must be accessible to the LucidWorks Search server. If authentication is required to access the repository, the data source must support the authentication type and the correct credentials supplied.
- The documents must be parseable, so the fields and content can be extracted.
- The specific data source settings must be configured to include the specific documents.

For example, if I have a file system with 100 PDF documents, each of which are OCR scans and 100Mb in size, the PDF documents: a) may not be parseable because OCR scans are images and, b) may exceed the maximum file size configured in the data source (the default is 10Mb). In this example, the files would be skipped by the crawler, which is not considered a serious exception and is generally only logged when the data source setting to "Log extra detail" is selected. Then the skipped files would be found in the log file with a format like this:

```
INFO filesystem.FileSystemCrawler - File <file-URL> exceeds the maximum size
specified for this data source. Skipping.
```

WARN No extractor for <file format>; Skipping: <document-URI>

Possible Errors

This information is provided to help you find the errors in the log file; precise troubleshooting requires information about the documents and system environment. If a document causes an error (besides being too large or the system being out of memory), it may be helpful to try to isolate it and try again to be sure it is the document causing the problem and not some other system error that may have occurred at the same time.

In each of the errors below, the document URI will be listed. For files this will be the path and filename, for websites it would be the URL, and for other data source types a base document URI will be configured based on how the data source is configured.

Exception

```
WARN Exception while crawling: <document-URI> <exception-with-stack-trace>
WARN Doc failed: <exception-with-stack-trace>
WARN Doc failed: <document-URI> - cause: <exception-cause-message>
```

PDF files are notorious for causing exceptions in their processing. These errors are not always fatal, but may cause all or part of the file to be skipped.

```
WARN util.PDFStreamEngine - java.io.IOException: Error: expected hex character and
not :32
WARN util.PDFStreamEngine - java.io.IOException: Error: expected the end of a
dictionary.
```

Out of memory

```
WARN File caused an Out of Memory Exception, skipping: <document-URI>
<exception-with-stack-trace>
WARN Doc failed: <exception-with-stack-trace>
WARN Doc failed: <document-URI> - cause: <OOM-exception-message>
```

SubCrawlerException

```
WARN Doc failed: <exception-with-stack-trace>
WARN Doc failed: <document-URI> - cause: <exception-message>
```

Unknown file type

```
WARN Doc failed: Could not find extractor: <document-URI>
```

In this case, this warning will be seen in the logs but will not be reported in the LucidWorksLogs collection.

I/O error

WARN IO Exception processing: <document-URI> <exception-with-stack-trace>

WARN Doc failed: <exception-with-stack-trace>

WARN Doc failed: <document-URI> - cause: <exception-message>

HTML/XML/XHTML parsing errors

WARN Doc failed: <exception-with-stack-trace>

WARN Doc failed: <document-URI> - cause: <exception-cause-message>

This is another case where a warning will be seen in the logs but will not be reported in the LucidWorksLogs collection.

Related Topics

- [System Directories and Logs](#)

Crawling Windows Shares with Access Control Lists

LucidWorks Search can crawl Windows Shares (SMB filesystems) and the Access Control Lists (ACLs) associated with shared files and directories. The ACL information can then be used to limit users' searches to the content they are permitted to access. This page describes how to configure using ACLs to control search results based on the user's permissions

As of LucidWorks Search v2.5, it's possible to configure ACL and Active Directory connections on a per-data source basis. This means that you can simply create a Windows Share data source with either the UI or the API, configure the connection to the Active Directory server, define if you want to trim results based on user authorizations, and then crawl the content.

When configuring the connection between LucidWorks Search and Active Directory, keep these requirements in mind:

- Credentials with READ and ACL READ permissions for accessing the Windows share. We recommend that you create a special user for this purpose.
- Credentials with read-only access to the Active Directory LDAP. This is used for search-time filtering, and we recommend that you create a special user for this purpose.

Permissions with Access Control Lists

The following model is implemented as a search filtering component by default:

Group READ Access	Subgroup READ Access	User READ Access	Search Result Returned?
o (permit)	o	o	o
o	× (deny)	o	×
o	o	×	×
o	×	×	×
×	o	o	×
×	×	o	×
×	o	×	×
×	×	×	×
o	- (not set)	o	o
o	o	-	o
o	-	-	o
-	o	o	o

-	-	o	o
-	o	-	o
-	-	-	x

To understand this table, read the rows left to right. For example, in the first row, we see that the user's main group, subgroup, and individual permissions all allow READ access to a shared resource, so the search result is returned. In the second row, we see that the user's main group and user's individual permissions allow READ access, but the user's subgroup's permissions do not, so no search result is returned to the user.

How SMB ACL Information Is Stored In The Index

For each file that is crawled through the SMB data source the `acl` field is populated with data that can be used at search time to filter the results so that only people that have been granted access at the user level or through group membership can see them. Two kinds of tokens are stored: Allow and Deny. The format used is as follows:

Allow:

WINA<SID>

Deny:

WIND<SID>

Where `SID` is the security identifier commonly used in Microsoft Windows systems. There are some well known SIDs that can be used in the `acl` field to make documents that are crawled through some other mechanism than by using SMB data source behave, from the `acl` `pow`, the same way as the crawled SMB content:

SID	Description
S-1-1-0	Everyone.
S-1-5-domain-500	A user account for the system administrator. By default, it is the only user account that is given full control over the system.
S-1-5-domain-512	Domain Admins: a global group whose members are authorized to administer the domain. By default, the Domain Admins group is a member of the Administrators group on all computers that have joined a domain.
S-1-5-domain-513	Domain Users.

Note that some of the listed SIDs contain a `domain` token. This means that the actual SIDs differ from system to system. To find out the SIDs for particular user in particular system you can use the information provided by the Windows command line tool `whoami` by executing command `whoami /all`.

You can populate the `acl` field in your documents with these Windows SIDs to make them searchable in LucidWorks Search. For example, if you wanted to make some documents available to "Everyone" you would populate the `acl` field with the `WINAS-1-1-0` token. If you wanted to make all docs from one data source available to everybody you can use the literal definitions in the data source configuration.


Related Topics

- Filtering API
- Search Handler Components API
- [LDAP Integration](#)

Indexing Binary Data Stored in a Database

This functionality is
not available with
LucidWorks Search
on AWS or Azure


The Database crawler in LucidWorks Search does not automatically discover and index binary data you may have stored in your database (such as PDF files). However, you can configure LucidWorks to recognize and extract the binary data correctly by modifying the data source configuration file (which does not exist until you create a JDBC data source).

 For detailed information about working with JDBC data sources, see [Create a New JDBC Data Source](#) or the Database Data Sources API.

After you have created a Database data source, you can find the configuration file in `$LWS_HOME/data/lucid.jdbc/datasources/id/conf/dataconfig.xml`. The ID in the path is the ID of the data source created. If you are familiar with Solr, you will recognize this file as a [Data Import Handler](#) configuration file.

Follow these steps to modify the configuration file:

1. Add a `name` attribute for the database containing your binary data to the `dataSource` entry.
2. Set the `convertType` attribute for the `dataSource` to `false`. This prevents LucidWorks from treating binary data as strings.
3. Add a `FieldStreamDataSource` to stream the binary data to the Tika entity processor.
4. Specify the `dataSource` name in the `root` entity.
5. Add an entity for your `FieldStreamDataSource` using the `TikaEntityProcessor` to take the binary data from the `FieldStreamDataSource`, parse it, and specify a field for storing the processed data.
6. Reload the Solr core to apply your configuration changes.

 After you have modified the data source configuration file you should not modify the data source from the LucidWorks Admin UI because LucidWorks will automatically overwrite the `convertType` attribute, and indexing for the modified data source will fail.

Example

In this example there is a MySQL database called `test` containing a table called `documents` that contains PDF data in a column called `binary_content`. When the data source is first created, the data source configuration file (in

`$LWS_HOME/data/lucid.jdbc/datasources/id/conf/dataconfig.xml`) looks like this:

```
<dataConfig>
  <dataSource autoCommit="true" batchSize="-1" convertType="true"
driver="com.mysql.jdbc.Driver" password="admin"
  url="jdbc:mysql://localhost/test" user="root"/>
  <document name="items">
    <entity name="root" preImportDeleteQuery="data_source:9" query="SELECT * FROM
documents"
      transformer="TemplateTransformer">
      <field column="data_source" template="9"/>
      <field column="data_source_type" template="Jdbc"/>
      <field column="data_source_name" template="MySQL"/>
    </entity>
  </document>
</dataConfig>
```

To modify this data configuration file, follow these steps:

1. Add the name attribute to the `dataSource` and set `convertType` to false:

```
<dataSource autoCommit="true" batchSize="-1" convertType="false"
driver="com.mysql.jdbc.Driver" password="admin"
  url="jdbc:mysql://localhost/test" user="root" name="test"/>
```

Specify another `dataSource` called `fieldReader` to handle the binary data:

```
<dataSource name="fieldReader" type="FieldStreamDataSource" />
```

2. Specify the data source for the root entity:

```
<entity name="root" preImportDeleteQuery="data_source:9" query="SELECT * FROM
documents"
  transformer="TemplateTransformer" dataSource="test">
```

3. Add an entity for the `fieldReader` data source specifying the `TikaEntityProcessor` and a `dataField` for storing the processed binary data:

```
<entity dataSource="fieldReader" processor="TikaEntityProcessor"
dataField="root.binary_content" format="text">
  <field column="text" name="body" />
</entity>
```

4. [Restart LucidWorks Search](#) to apply your configuration changes.

For this example, the final configuration file looks like this:

```
<dataConfig>
  <dataSource autoCommit="true" batchSize="-1" convertType="false"
driver="com.mysql.jdbc.Driver" password="admin"
  url="jdbc:mysql://localhost/test" user="root" name="test"/>
  <dataSource name="fieldReader" type="FieldStreamDataSource" />
  <document name="items">
    <entity name="root" preImportDeleteQuery="data_source:9" query="SELECT * FROM
documents"
      transformer="TemplateTransformer"
      dataSource="test">
      <field column="data_source" template="9"/>
      <field column="data_source_type" template="Jdbc"/>
      <field column="data_source_name" template="MySQL"/>
      <entity dataSource="fieldReader" processor="TikaEntityProcessor"
dataField="root.binary_content" format="text">
        <field column="text" name="body" />
      </entity>
    </entity>
  </document>
</dataConfig>
```

Related Topics

- [Create a New JDBC Data Source](#)
- [Database Data Sources API](#)
- [Data Import Handler](#)

Using the High Volume Crawlers

The High Volume HDFS (HV-HDFS) Crawlers are MapReduce-enabled crawlers designed to leverage the scaling qualities of [Apache Hadoop](#) and [MapR](#) while indexing content into LucidWorks Search. In conjunction with LucidWorks' usage of [SolrCloud](#), applications should be able to meet their large scale indexing and search requirements.

To achieve this, the high volume crawlers consist of a series of MapReduce-enabled Jobs to convert raw content into documents that can be indexed into LucidWorks which in turn relies on the [Behemoth project](#) (specifically, the [LWE](#) fork/branch of Behemoth hosted on Github) for MapReduce-ready document conversion via [Apache Tika](#) and writing of documents to LucidWorks.

The high volume crawlers are currently marked as "Early Access" and are subject to changes in how they works in future release.

System Requirements

- Apache Hadoop or MapR. We've tested with Hadoop v1 and v2 and MapR 2.1.2. Other versions may also work, but we recommend thorough testing for compatibility.
- LucidWorks running in [SolrCloud mode](#).

Please note, explanation of setting up Hadoop or MapR is beyond the scope of this document. We recommend reading one of the many tutorials found online or one of the books on Hadoop or MapR.

Using a Local Hadoop Instance

LucidWorks has been tested with Hadoop v1 and v2. The HV-HDFS and HDFS data sources can use the version of Hadoop bundled with LucidWorks Search (v2.0.5-alpha), or an external Hadoop of either version 1 or version 2 can be used by specifying a parameter in the system `master.conf`. The parameter varies depending on the Hadoop version being used: enter `-Dhadoop` if using Hadoop v1 or `-Dhadoop2` for Hadoop v2. The value entered should be the path to the Hadoop client libraries, and complete (non-relative) paths should be used (such as `-Dhadoop2=/path/to/Hadoop`).

Note that this system parameter will apply to all HDFS data sources.

Special Requirements for MapR


- Modify the default DirectoryFactory. If you intend to use the MapR High-Volume data source, you should use Solr's `NIOFSDirectoryFactory` instead of the default `SimpleFSDirectory`. You can change this by editing the `lwecore.jvm.params` in `$LWS_HOME/conf/master.conf` and adding `"-Dsolr.directoryFactory=solr.NIOFSDirectoryFactory"` to the end of the settings already there. More information about the `NIOFSDirectoryFactory` is available in the [Lucene javadocs documentation](#).

- **MapR Client.** The MapR client must be installed at a filesystem location accessible by the LucidWorks Connector component. For information about the MapR client, please see the MapR documentation [Setting Up the Client](#). The Connector component looks for the client libraries in `/opt/mapr` by default, but the location can be modified by editing the `lweconnectors.jvm.params` in `$LWS_HOME/conf/master.conf`. Find the setting `-Dmapr.home` and modify the path as needed. On Windows, you will need to include the drive (i.e., `c:` or `d:`) and also use two backslashes following the drive letter, as in `c:\\opt\\mapr`.

Using High Volume Crawlers in LucidWorks

Once Hadoop or MapR and LucidWorks are ready, configure a data source within LucidWorks, either with the [High Volume HDFS data source type](#) or the [MapR High Volume Crawler](#) in the Admin UI or using the Data Sources API.


If you do not see the MapR High Volume Crawler in the UI, or get an error from the API of "unknown crawler type lucid.map.reduce.maprfs", it is likely because the MapR Client is not installed or not accessible to the Connectors component. Please see the information in the System Requirements section for [more details about the MapR Client](#).

 Unlike other crawlers in LucidWorks Search, these crawlers currently have no way of tracking which content is new, updated, or deleted. Thus, all content found is reported as "new" with each crawl. It is also not possible to configure [batch operations](#) with the high-volume data source types.

How it Works

The high volume crawlers consist of three stages designed to take in raw content and output results to LucidWorks Search. These stages are:


1. Create one or more SequenceFiles from the raw content. This can be done in one of two ways:
 1. If the source files are available in a shared Hadoop filesystem, prepare a list of source files and their locations as a SequenceFile. The raw contents of each file are not processed until step 2.
 2. If the source files are not available, prepare a list of source files and the raw content, stored as a BehemothDocument. This process is currently done sequentially and can take a significant amount of time if there is a large number of documents and/or if they are very large.
2. Run a MapReduce job to extract text and metadata from the raw content using Apache Tika. This is similar to the LucidWorks approach of extracting content from crawled documents, except it is done with MapReduce.
3. Run a MapReduce job to send the extracted content from HDFS to LucidWorks using the SolrJ client. This implementation works with SolrJ's CloudServer Java client which is aware of where LucidWorks is running via Zookeeper.

 The processing approach is currently all or nothing when it comes to ingesting the raw content and all 3 steps must be completed each time, regardless of whether the raw content hasn't changed. Future versions may allow the crawler to restart from the SequenceFile conversion process.

Permission Issues

Using either Hadoop or MapR, you will need to be aware of the way Hadoop and systems based on Hadoop (such as MapR) handle permissions for services that communicate with other nodes.

Hadoop and MapR services execute under specific user credentials: a quadruplet consisting of user name, group name, numeric user id, numeric group id. Installations that follow the manual usually use user 'mapr' and group 'mapr', with numeric ids assigned by the operating system (e.g., uid=1000, gid=20). When the system is configured to enforce user permissions (which is the default in MapR), any client that connects to Hadoop or MapR services has to use a quadruplet that exists on the MapR server. This means that ALL values in this quadruplet must be equal between the client and the server, i.e., an account with the same user, group, uid, and gid must exist on both client and server machines.

 While it's easy to create a user with a given name and group name, it's less obvious to casual users how to create an account with exactly the same numeric id-s. On POSIX systems (Linux and Mac) it's possible to do so, on Windows it's probably not possible. For this reason there's a section of code in Hadoop and MapR to "spoof" user ids on Windows, using the following properties:

- `hadoop.spoof.user`: boolean, when **true** then spoofing will be attempted
- `hadoop.spoofed.user.username`: name of the user account to spoof
- `hadoop.spoofed.user.groupname`: group name of the user account to spoof
- `hadoop.spoofed.user.uid`: numeric user id of the user account to spoof
- `hadoop.spoofed.user.gid`: numeric group id of the user account to spoof

These properties will be used ONLY on Windows. Users on other operating systems will have to create a real account with matching identifiers.

When a client attempts to access a resource on Hadoop or MapR filesystems (or the MapR JobTracker, which also uses this authentication method) it sends its credentials, which are looked up on the server, and if an exactly matching record is found then those local permissions will be used to determine read/write access. If no such account is found then the user is treated as "other" in the sense of the permission model.

This means that the crawlers for the HDFS and MapR data sources should be able to crawl Hadoop or MapR filesystems without any authentication, as long as there is a read (and execute for directories) access for "other" users granted on the target resources. Authenticated users will be able to access resources owned by their equivalent account.

However, with the High-Volume HDFS and MapR High Volume data sources require write access to a `/tmp` directory to use as a working directory. In many cases, this directory does not exist, or if it does, it doesn't have write access to "other" (not authenticated) users. Therefore users of these data sources should make sure that there is a `/tmp` directory on the target filesystem that is writable using their local user credentials, be it a recognized user, group, or "other". If a local user is recognized by the server then it's enough to create a `/tmp` directory that is owned by that user. If there is no such user, then the `/tmp` directory must be modified to have write permissions for "other" users. The working directory can be modified to be another directory that can be used for temporary working storage that has the correct permissions.

Differences from Other Hadoop Crawlers in LucidWorks

While the HV-HDFS, MapR High Volume, Hadoop File System (HDFS) and Hadoop File System over S3 (S3H) crawlers all use Hadoop to access Hadoop's distributed file system, there is a big difference in how they utilize those resources. The HDFS and S3H data sources are designed to be polite and crawl through the content stored in HDFS just as if they were crawling a web site or any other file system.

The HV-HDFS and MapR High Volume crawlers, on the other hand, are designed to take full advantage of the scaling abilities of the MapReduce architecture. Thus, it runs jobs using all of the nodes available in the cluster just like any other MapReduce job. This has significant ramifications for performance since it is designed to move a lot of content, in parallel, as fast as possible (depending on the system's capabilities), from its raw state to the LucidWorks Search index. Thus, you will need to design your LucidWorks Search SolrCloud implementation accordingly and make sure to provision the appropriate number of nodes.

Conversion to SequenceFiles

The first step of the crawl process converts the input content into a SequenceFile. In order to do this, the entire contents of that file must be read into memory so that it can be written out as a BehemothDocument in the SequenceFile. Thus, you should be careful to ensure that the system does not load into memory a file that is larger than the Java heap size of the process. In certain cases, Behemoth can work with existing files such as SequenceFiles to convert them to Behemoth SequenceFiles. Contact LucidWorks for possible alternative approaches.

Example: Indexing Shakespeare with MapReduce

The following steps demonstrate indexing the complete works of Shakespeare using the LucidWorks Search HV-HDFS crawler.

Prepare the Content

1. Download the data into a temporary directory.
 1. `cd /tmp`
 2. `mkdir shakespeare`
 3. `wget http://www.it.usyd.edu.au/~matty/Shakespeare/shakespeare.tar.gz`
2. Unpack the archive with `tar -xf shakespeare.tar.gz -C shakespeare`
3. Load the data to Hadoop with `<PATH TO HADOOP>/bin/hadoop fs -put shakespeare ./shakespeare`

Setup LucidWorks Search

Start LucidWorks Search in SolrCloud mode and make note of where Zookeeper is running. See the section on [Using SolrCloud in LucidWorks](#) for more information on how to start in SolrCloud mode.

Setup the Data Source and Run

Create a new data source using either the Admin UI or the Data Source API, as described [above](#). The "path" would be the location of the Hadoop NameNode, such as, `hdfs://lucidserver:54310/user/lucidworks/shakespeare`.

When using MapR High Volume, you can leverage CLDBs (if you're using them) to crawl all the nodes of your cluster. See the section [Create a New MapR_HV Data Source](#) for details.

Once the data source is created, you can use the Hadoop or MapR UI to track the progress of the various MapReduce jobs. You can also inspect your specified work path to see the intermediate files using the Hadoop filesystem commands (e.g., `hadoop fs -ls`).

Related Topics

- [Using SolrCloud in LucidWorks](#)
- [Apache Hadoop](#)
- [Behemoth](#)
- [Scaling Solr Indexing With SolrCloud, Hadoop and Behemoth](#)

Suggestions for External Data Sources

In some cases, it may not be possible to use the crawlers included with LucidWorks Search to index content, such as an email archive or a Web repository that's best accessed via an API. Instead, another process may be possible, such as using [SolrJ](#), to feed documents directly to Solr. In that situation, LucidWorks would not normally know about the documents and would not be able to include information about the data source in facets or display statistical data about the data source in the Admin UI.

Fortunately, there is a way to create an 'external' data source to add fields to the document so LucidWorks will treat the documents the same as documents found via the embedded crawlers. The data source can be created either via the Sources screen in the [Admin UI](#) or with the Data Sources API.

The 'external' data source type is different from the other data source types in that it is the only one that uses a "push" mechanism to push content into LucidWorks (and, by extension, Solr), while the other data source types use a "pull" model to go and get content for processing. Because the content is being pushed from an external process, these suggestions will ensure that they are processed consistently by LucidWorks Search.

Add the `fm.ds` Parameter to the Push Request

External data sources support field mapping in the same way all other data source types do (by specifying the mapping with the Data Sources API or via the Admin UI). The `fm.ds` parameter in the request allows LucidWorks to know which data source's field mapping rules to apply to the content as it is being processed. Without this parameter in the request, the default field mapping will be applied, even if the field mapping has been customized for the data source.

The default search UI included with LucidWorks relies on the `title` and `body` fields being populated in order to display information about results to users; the `author` and `lastModified` fields are also used for display and faceting. If your custom search UI uses these or other fields for display of results, it's recommended that the documents pushed directly to Solr include content in the those fields for a consistent user experience.

The value of the `fm.ds` parameter should match the ID of the data source, which can be found with either the Data Sources API (by reviewing the full list of data sources) or via the Admin UI (by inspecting the URL of the Edit Settings screen for the specific data source ID). If the ID supplied does not match an existing data source ID, an error will be returned and the documents will not be loaded.

It is also possible to supply `'-1'` for the `fm.ds` ID. In cases where there is only one external type of data source, the fields for that data source will be filled in to the documents and the mapping for that data source will be applied. If there are zero external data sources or more than one external data source, an error will be returned and the documents will not be loaded.

Add `lucidworks_fields` to Incoming Content

When LucidWorks crawlers acquire content, certain fields related to the data source are added to each document to help identify the documents as belonging to the data source for use in statistics, faceting, and document deletion (if necessary). This is done via an attribute called `lucidworks_fields` (which is shown as "Create LucidWorks fields" in the Edit Mapping screen of the Admin UI). The default for this attribute is "true", which means the fields will be added to all incoming documents, so usually no editing is required to add these fields as long as the `fm.ds` parameter has been added to the update request.

The fields added to each document are from the data source, but have different names. This table shows the relationship between the data source attribute name and the fields added to documents:

Data Source Attribute	Field Name (in <code>schema.xml</code>)
id	data_source
type	data_source_type
name	data_source_name

Schedule the Data Source with the callback Attribute

The external data source can be scheduled in the same ways as any other data source to periodically update, delete or add new documents. The `callback` attribute must be set in order for the schedule to work correctly. This attribute requires LucidWorks to issue an HTTP GET request to trigger the push when a job is started, either via the scheduler or manually via API or the UI. This call occurs just after LucidWorks updates the field mapping, so if the mapping is modified between schedules incoming documents get the new mapping.

Examples

Using the Data Sources API, a new data source could be created with these settings:

```
curl -X post -H 'Content-type: application/json' -d '{"name":"Test External #1","type":"external","crawler":"lucid.external","source_type":"Raw SolrXML","source":"Solr update"}' http://localhost:8888/api/collections/collection1/datasources
```

The output of this command would be as follows:

```
{
  "commit_on_finish":true,
  "verify_access":true,
  "indexing":true,
  "source_type":"Raw SolrXML",
  "collection":"collection1",
  "type":"external",
  "crawler":"lucid.external",
  "id":3,
  "category":"External",
  "source":"Solr update",
  "name":"Test External #1",
  "parsing":true,
  "commit_within":900000,
  "caching":false,
  "max_docs":-1
}
```

Then a document such as this could be added directly to Solr:

```
curl -H 'Content-type: text/xml' --data-binary '<add> <doc> <field
name="id">testdoc</field> <field name="body">test</field> </doc> </add>'
http://localhost:8888/solr/collection1/update?fm.ds=3&commit=true
```

Here is an example document using SolrJ:

```
...
String dsId = "3";
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id", "1234");
doc.addField("body", "test");

SolrServer server = new
CommonsHttpSolrServer("http://localhost:8888/solr/collection1");
UpdateRequest req = new UpdateRequest();
req.setParam("fm.ds", dsId);
req.add(doc);
req.process(server);
```

Related Topics

* [Solr Direct Access](#)

- [Indexing and Basic Data Operations](#) from the Apache Solr Reference Guide.

Indexing Documents Directly to Solr

Solr provides many ways to index content, and these can be used in addition to or instead of the crawlers built into LucidWorks Search. Solr includes several approaches to indexing content:

- Solr can index XML (in a specific Solr format), CSV files and JSON formats natively
- Solr Cell (Content Extraction Library) uses Tika to extract documents from a variety of sources
- SolrJ is used by many to connect their Java applications to Solr for indexing and also querying document once they've been indexed
- The DataImportHandler (DIH) provides access to structured data in relational databases (the Database data source in LucidWorks uses DIH under the hood)
- Crawling can be done with Nutch and then pushed into Solr

This page provides a brief overview of how to index content into Solr; for more information, including details of the options mentioned above, please see the Solr Reference Guide section on [Indexing and Basic Data Operations](#).

Solr and the LucidWorks Admin UI

If you push documents directly to Solr without using LucidWorks Search data sources, the LucidWorks Admin UI will be unable to display statistical information about those documents. This is because documents crawled via LucidWorks Search contain a field that includes the data source ID, and the data source ID is used by the Admin UI to display information such as the number of documents in the index for that data source, and to know which crawl statistics to display.

The LucidWorks data source type "external" would allow you to integrate documents pushed directly to Solr with documents indexed from the crawlers and get statistics such as number of documents per data source in the Admin UI. In addition, the external data source also allows using LucidWorks data source field mapping functionality. For more information, see [Suggestions for External Data Sources](#); the information contained below is still valid, but would be slightly modified when using the "External" approach.

Indexing Solr XML

One way to integrate LucidWorks with a custom data source is to dump the data from that data source into XML files formatted in this way, and index them as a Solr XML data source. LucidWorks has built-in support for indexing a directory tree of Solr XML files and scheduling periodic re-indexing. Alternatively, the XML files can easily be posted into LucidWorks and Solr externally using curl, the REST API, or other tools that can HTTP POST, like this:

```
curl http://localhost:8888/solr/collection1/update --data-binary @filename.xml -H
'Content-type:text/xml; charset=utf-8'
```

Solr natively digests a simple XML structure like this:


```
<add>
  <doc>
    <field name="fieldname1">field valueA</field>
    <field name="fieldname2">field valueB</field>
  </doc>
  <doc>
    <field name="fieldname3">multivalue1</field>
    <field name="fieldname3">multivalue2</field>
  </doc>
</add>
```

The `<add>` structure supports multiple `<doc>` declarations and each `<doc>` supports multiple `<field>` declarations. Fields can be multi- or single-valued, depending on the `schema.xml` configuration. The LucidWorks Search [Fields screens](#) provide a handy user interface for managing field properties, including the multivalued setting.

Solr's XML format can perform other operations including deleting documents from the index, committing pending operations, and optimizing an index (a housekeeping operation). For more information on these operations, as well as adding documents, refer to [Solr's Update XML Messages](#).

Indexing Column (Comma) Delimited Data

The following section uses an example to illustrate how to index delimited text with LucidWorks.

1. Save the following simple comma-separated data as `sample_data.text`:

```
id,title,categories
1,Example Title,"category1,category2"
2,Another Record Example Title,"category2,category3"
```

2. Configure the index schema using the Fields editor in the Admin UI as follows:
 - At the bottom of the page, click **Add new field** to get a blank field form
 - Add a new field with the following settings:
 - Name: categories
 - Type: string
 - Stored: checked
 - Multi-valued: checked
 - Short Field Boost: none
 - Search by Default: checked
 - Include in Results: checked
 - Facet: checked
3. Save and apply those settings.

4. Now index the CSV data from the command-line using curl:

```
curl
"http://localhost:8888/solr/collection1/update/csv?commit=true&f.categories.split=true"
--data-binary @sample_data.txt -H 'Content-type:text/plain; charset=utf-8'
```

5. You can also make the file pipe-delimited, like this:

```
id|title|categories
3|Three|category3
4|Four|category4,category5
```

And then you can index using this command:

```
curl
"http://localhost:8888/solr/collection1/update/csv?commit=true&f.categories.split=true"
--data-binary @pipe.txt -H 'Contenttype:text/plain; charset=utf-8'
```

For a full description of all CSV options, see the [Solr UpdateCSV](#) documentation.

Related Topics

- [Suggestions for External Data Sources](#)

From our Apache Solr Reference Guide:

- [Indexing and Basic Data Operations](#)
- [Using SolrJ](#)

Integrating Nutch

LucidWorks Search includes support for "external" data sources (also known as "push crawlers"). While the built-in LucidWorks crawlers use the "pull" model (meaning that LucidWorks initiates the crawl and actively discovers new or updated resources), push crawlers are external processes that manage the discovery and sending of new and updated documents for indexing outside of the LucidWorks crawler framework.

Apache Nutch is a framework for building and running large-scale Web crawling using Hadoop map-reduce clusters (see <http://nutch.apache.org/> for more information). Recent releases of Nutch rely on Solr for indexing and searching. From the point of view of LucidWorks, Nutch can be integrated as an "external" or "push" crawler.

The following sections describe step-by-step how to integrate a [Nutch 1.4](#) crawler (or Nutch [trunk](#)) with LucidWorks.

Solr indexer

Nutch comes with a tool for map-reduce indexing to Solr called `SolrIndexer`. From the command-line, this tool is invoked like this:

```
nutch solrindex http://localhost:8983/solr/collection1 db -linkdb linkdb [-params  
k1=v1,k2=v2] segment1 segment2 [...]
```



Support for the `-params` option exists in Nutch trunk, post 1.4 release, or if you apply the patch found in [NUTCH-1212](#)).

Field mapping in Nutch

Nutch uses indexing plugins to construct the outgoing documents, and these plugins add various fields with various names. These field names do not necessarily match the default LucidWorks `schema.xml` for a collection. Nutch provides a limited facility to adjust these names (see `$nutch_home/conf/solrindex-mapping.xml`). This field mapping facility is often enough in simple cases to re-map field names so that they match the LucidWorks schema.

However, this solution has some drawbacks:

- This mapping is static for all indexing jobs that use the same job file (or the same `conf` directory in the case of a non-distributed Nutch installation) and changing it requires rebuilding of the job file, which can be cumbersome.
- There is no easy way to add fields that are useful for managing documents in LucidWorks (such as `data_source_type`, `data_source_name` or `data_source`), short of implementing a new Nutch indexing plugin.

- the field mapping in `solrindex-mapping.xml` cannot be managed from the LucidWorks Admin UI.

Fortunately, there is a better solution to this problem which is to use the field mapping functionality in LucidWorks, defined as part of the External data source type definition, in combination with the `-params` option for `SolrIndexer`.

Field mapping in LucidWorks

External processes that submit documents to LucidWorks can be integrated using the External data source type. When you define a new data source in LucidWorks, one of its properties is `field_mapping`. With the Data Sources API, the JSON serialization looks similar to this:

```
...
"mapping": {
  "datasource_field": "data_source",
  "default_field": null,
  "dynamic_field": "attr",
  "literals": {},
  "lucidworks_fields": true,
  "mappings": {
    "acl": "acl",
    "author": "author",
    "batch_id": "batch_id",
    "content": "body",
    "content-encoding": "characterSet",
    "content-length": "fileSize",
    "content-type": "mimeType",
    "contentcreated": "dateCreated",
    "contentlastmodified": "lastModified",
    ...
  },
  "multi_val": {
    "acl": true,
    "author": true,
    "body": false,
    "dateCreated": false,
    "description": false,
    "fileSize": false,
    "mimeType": false,
    "title": false
  },
  "types": {
    "date": "DATE",
    "datecreated": "DATE",
    "filesize": "LONG",
    "lastmodified": "DATE"
  },
  "unique_key": "url",
  "verify_schema": true
},
...
```

The LucidWorks Admin UI includes a page for each data source to edit field mapping for that data source which is where you can define, for example, that "content" should be mapped to "body", or that you allow only a single value for "title", etc.

In particular, you can define what is the name of the "uniqueKey" field in the incoming documents. If Nutch produces documents that use "url" as their unique identifier, then you would specify "uniqueKey": "url". If "verify_schema" is set to "true" then LucidWorks will automatically define a mapping from "url" to whatever the current "uniqueKey" field is in the Solr schema for the target collection.

Once the external data source is defined (or updated) LucidWorks sends the serialized field mapping to the FieldMappingUpdateProcessor, which is a part of the "lucid-update-chain". This update processor receives the field mapping definition, and stores it in memory under a specified data source id. This field mapping is then updated each time a user makes some modifications to the data source definition, either via the Admin UI or using the REST API.

From this point, whenever an update request is received from an external process and it goes through this update chain, the update processor looks for a Solr parameter "fm.ds", which indicates the data source ID. If this parameter is present, and matches an existing defined mapping, then the documents in the update request are put through the FieldMappingUpdateProcessor, which re-maps field names, adjusts field multiplicity and adds LucidWorks-specific field names and values (which, among others, help to manage documents using the LucidWorks Admin UI).

Putting it all together

Now that we know how the field mapping is configured and processed in LucidWorks we can make sure that Nutch SolrIndexer uses the correct parameters, so that the correct field mapping is applied in LucidWorks to documents arriving from Nutch. Let's say that our external data source in LucidWorks has a data source id "4", we want to add the documents to "collection1" and our LucidWorks instance is running on a host "lucidworks.io:8888". Then the command-line parameters to SolrIndexer would look like this:

```
nutch solrindex http://lucidworks.io:8888/solr/collection1 db -linkdb linkdb -params
'update.chain=lucid-update-chain&fm.ds=4' segment1 segment2 [...]
```

As you can see, we are using the target collection's URL, and we specify "fm.ds=4" parameter that determines what field mapping needs to be applied to the incoming documents. Just in case, we explicitly set the update chain in case "lucid-update-chain" is not the default one (which it is in an out-of-the-box installation of LucidWorks). Please note that the `-params` option uses a URL-like syntax for passing Solr parameters, and since ampersand is usually a special shell character we had to enclose the `-params` string in single quotes to prevent the shell from interpreting it.

Summary

Nutch and LucidWorks form a powerful combination. Nutch is a robust crawling platform that can easily crawl thousands of pages per second while LucidWorks offers a scalable and robust indexing and search platform.

The way to use the two together is simply to:

- Define an "external" data source in LucidWorks, and adjust its field mapping to properly map the default Nutch field names to the ones that make sense in the current LucidWorks schema (e.g., "uniqueKey":"url", "content":"body", etc.). An external data source can be created by choosing the "External" type in the Sources page of the Admin UI or with the Data Sources API, specifying "lucid.external" for the `crawler` and "external" for the `type`.

- Start the Nutch SolrIndexer job with the additional -params option that specifies the data source id of the "external" data source defined in LucidWorks.

Related Topics

- [Suggestions for External Data Sources](#)
- [Apache Nutch homepage](#)

Processing Documents in Batches

By default, LucidWorks Search will crawl as much content as it can (within limits set on the data source), parse the documents to extract fields, and finally index the documents in one seamless step. However, there may be times when you would like to do some processing on the documents before indexing them, perhaps to add metadata or to modify data in specific fields. In that case, it is possible to only crawl the content and save it in a batch for later parsing and/or indexing. This is called Batch Processing and allows you to separate the fetching data from the process of parsing the rich formats (such as PDFs, Microsoft Office documents, and so on), as well as the process of indexing the parsed content in Solr.

How a Batch is Constructed

Batches consist of the following two parts:

- a container with raw documents, and the protocol-level metadata per document
- a container with parsed documents, ready to be indexed.

The exact format of this storage is specific to a crawler controller implementation. Currently a simple file-based store is used, with a binary format for the raw content part and a JSON format for the parsed documents. The first container is created during the fetching phase, and the second container is created during the parsing phase. A new round of fetching creates a new batch if one or more of the parameters described above requires it.

Steps to Configure Batch Crawling

It's not possible to configure Batch Crawling with the LucidWorks Search Admin UI. To work with batches and batch jobs, use the Batch Operations API. The basic workflow is as follows:

1. Create a data source using the [Admin UI](#) or Data Sources API. Don't start crawling yet.
2. Configure the data source to be saved as a batch by setting the `indexing` parameter to `false` using the Data Sources API. You can also set the `caching` and `indexing` parameters as described below.
3. Start the crawl and let it finish.
4. Get the `batch_id` for the data source using the Batch Operations API call: `GET http://localhost:8888/api/collections/collection1/batches`.
5. Using the Batch Operations API, start the batch job for your data source using the `batch_id` obtained in the previous step:

```
PUT http://localhost:8888/api/collections/collection1/batches/crawler/job/  
batch_id.
```

More about the Data Source Settings

To instruct LucidWorks Search not to parse or index the crawled documents, set the `indexing` parameter of a data source to `false` using the Data Sources API. You can also set the `parsing` and `caching` parameters to true or false, depending on your needs. Batch crawling attributes for data sources are as follows:

Key	Type	Default	Description
<code>parsing</code>	boolean	true	If true, the raw content fetched from remote repositories is immediately parsed in order to extract the plain text and metadata. If false, the content is not parsed: it is stored in a new batch with its protocol-level metadata. New batches are created during each crawl run as needed.
<code>caching</code>	boolean	false	If true, the raw content is stored in a batch even if immediate parsing and/or indexing is requested. You can use this to preserve the intermediate data in case of crawling or indexing failure, or in cases where full re-indexing is needed and you would like to avoid fetching the raw content again.
<code>indexing</code>	boolean	true	If true, the parsed content is sent to Solr for indexing. If false, the parsed document is not indexed: it is stored in a batch (either a newly created one, or the one where the corresponding raw content was stored). Set this attribute to <code>false</code> to enable batch crawling.

- ✓ When you configure a data source to process documents as a batch, information about crawl attempts will display in the Admin UI for that data source (even though you cannot configure the batch parameters via the UI). So, you can use the Data Sources API to enable `caching` and/or disable `indexing`, and initiate the crawl through the Admin UI. The UI will show the number of documents found, updated, deleted, etc.

Not all crawler controllers support all batch processing operations. For example, the Aperture crawler (`lucid.aperture`) does not support raw content storage: it behaves as if the "parsing" parameter is always `true` and caching is always `false`. Also, the MapR High Volume Data Sources and High-Volume HDFS Data Sources do **not support** any kind of batch processing.

You can also use the Batch Operations to get the status of or stop running batch jobs as well as delete batches and batch jobs.

Related Topics

- Batch Operations
- Data Sources

Query and Search Configuration

Once your content is in the index, you and your users will want to query the index to find the documents they need. This section covers the options and settings to optimize the search experience for users.

First, there's an overview of how searching works in the section [Overview of Query Processing](#).

A few features make it easy for users to find documents: [Enterprise Alerts](#) allow them to get email notifications when new documents are added to the index; [Spell Check](#) corrects errors in terms they've entered; [Auto-Complete of User Queries](#) makes suggestions for valid terms while they type, and [Synonyms and Stop Words](#) allows use of similar terms and very common words to improve the search experience.

While LucidWorks Search includes a Search UI, it's meant to be used during development and not for a production application. The section [Getting Search Results](#) describes in detail how to query the LucidWorks Search index, and what responses look like, for use while designing your own search application customized for your needs.

You may have need to improve the results your users see. The [Click Scoring Relevance Framework](#) provides a way to boost documents that other users have already clicked on for the same query, with the theory that if other users found it useful, you might too.

If you have serious business needs for including very specific rules in response to certain queries (or all queries), the section [Business Rules Integration](#) describes how to plug in those rules with LucidWorks Search.

Overview of Query Processing

The goal for any search application is to return the correct document while allowing a user to enter a query however they want. The query may be in the form of keywords, a natural language question, or snippets of documents. Advanced queries may be (or may include) date ranges, Boolean operations, searches on specific document fields, or proximity information to define how close (or how far apart) terms should be to each other.

Features like spell check and auto-complete can help prompt users to enter terms that are more likely to retrieve results. In LucidWorks Search, spell check provides suggestions for terms close to the user's terms, but which definitely exist in the index (that is the default implementation; a dictionary could be used instead). Auto-complete also provides suggestions based on terms in the index, but does so while the user is typing their query, providing real-time feedback to the user. More details are available in the sections [Spell Check](#) and [Auto-Complete of User Queries](#).

Matching the User's Query to Documents

Once the user hits enter, search engines take the query and transform it to find the best results. The section [Getting Search Results](#) describes how your search application should send the user's query to LucidWorks Search, and how the response will be formatted.

[Synonyms](#) of the terms entered may be applied to expand the number of possible document matches (such as looking for "attorney" when a user enters "lawyer"). If terms are stripped of punctuation and capital letters during indexing, a similar process should also be applied to the user query to ensure matches in the index. In LucidWorks Search, much of this is pre-configured but could be modified if needed.

The system then tries to match the user's transformed terms to terms in documents in the [index](#). Once it finds documents, it puts the list of matching documents into some order. They might be ordered by date, by entry to the index, or, most commonly, by *relevance*, which is an order based on which the system thinks are best for the query entered.

Relevance ranking is one of the most complex components of a search engine, and this guide covers the topic in more detail later (see [Understanding and Improving Relevance](#)). Most queries are very short (one to three words) and that is usually not enough information to know the user's full intention. To compensate for this, several techniques may be employed such as boosting based on the number of times the user's search terms appear in a document or boosting based on the location of the user's search terms in a document (in the title, at the beginning, etc.). Some approaches may drop very small words like "of", or "the" (also called *stop words*), so they don't unduly influence the term calculations.

Other techniques used in relevance ranking include considering the date of the item (documents that are more recent may be considered more relevant to some users) or where the term matches occur (words in the title of the document may be more relevant than words at the end). LucidWorks Search includes the option to use [Click Scoring](#), which uses information about the documents other users have selected as a factor when calculating relevance.

Search Results

Once the system has compiled a list of matching documents, they need to be presented to the user with enough information to help them decide which documents are best. First, the documents should be sorted in some way: the most common is by how well the documents match the query (relevance), but date may also be preferred, or another field such as author or manufacturer. Some snippet of the document should be used to help users figure out if the document is a match, such as title, author and date. The first few sentences, or a few sentences around the highlighted occurrence of the user's search term, are also helpful to give the user some context for why each document was selected as a match.

Document clustering, also called faceting, can help users select from a large list of results. Facets are documents grouped together by some common element such as author, type, or subject and are usually displayed with the number of results that can be found in each group. Providing facets allows users to "drill down" or further restrict their results and find the documents they are looking for.

Users may also benefit from tools to expand their queries without providing additional search terms. A "find similar" option allows users to request documents that are similar to one they consider almost right. Explicit or automatic feedback allows users to resubmit their search with terms pulled from documents that are considered near matches, in hopes of getting more or better matches. In LucidWorks Search, [unsupervised feedback](#) can be enabled, which automatically takes the top documents from the preceding results and pulls important terms from them to use with the user's original query.

Some queries are run on a periodic basis (daily, weekly, etc.). LucidWorks Search includes a feature to allow users to save their queries and the system will run them at defined intervals and send a notification if new documents have been added that match their query. This feature is called [Enterprise Alerts](#).

Result lists may need to be limited to only documents that a user has access to view. LucidWorks Search has several options for doing this, described in the section [Securing LucidWorks](#).

Getting Search Results

LucidWorks Search includes a default search interface that is designed to be used during development to evaluate and test the performance of crawler and index configuration. At this time, there are no options to customize the default Search UI because we expect that you will prefer your own designs and options specifically tailored to your audience.

What follows is some information about how to start working with search results in LucidWorks Search.

LucidWorks is built upon Solr and supports it natively. While LucidWorks includes a REST API for many administrative functions (like creating data sources, updating fields, etc.), there is no LucidWorks-specific API for search results. In order to get results from LucidWorks, you'll need to learn a little Solr syntax. To help you with this, you may find it helpful to review LucidWorks' free [Apache Solr Reference Guide](#), particularly the section on [Searching](#).

This page is an introduction to Solr searching.

You should also look at these sections:

- [Constructing Solr Queries](#)
- [Solr Query Responses](#)

Basics of Searching

Searching LucidWorks Search makes a direct connection to Solr, which processes queries with a **request handler**. The request handler defines the logic to be used for processing the query. Solr supports several different request handlers, and LucidWorks includes a special Solr search request handler called `/lucid`. Details about this special request handler are in the section [Lucid Query Parser](#).

The `/lucid` handler is pre-selected as the default, but could be changed to another request handler by editing `solrconfig.xml` for the collection. The simplest way to do this is to change the `defType` parameter from "lucid" to "edismax", "dismax" or a custom parser you've created.

Request Handlers

Each request handler has several settings pre-configured, but these can be overridden for an individual query by the client application. In some cases, this may adversely affect the expected search results, so care should be taken when overriding some parameters.

To process a query, a request handler calls a **query parser**, which interprets the terms and parameters of a query. The query parser understands the terms the user entered (the actual words), any parameters entered for fine-tuning the query (such as instructions to search a specific field for the terms, to boost terms found in specific fields to rank them higher in results, and to interpret the syntax for advanced queries including ranges or boolean operators, etc.), and any parameters for controlling the presentation of the response (such as the order of results or the fields of a document to be returned). LucidWorks has created its own query parser that is used by default, but any other Solr query parser could also be used (the two most popular are `DisMax` and `ExtendedDisMax`).

The request handler also likely has defined many parameters for faceting, spell check, autocomplete, highlighting, security settings and so on. The `/lucid` request handler has enabled and defined each of those components by default; with other request handlers those may need to be defined in `solrconfig.xml` or defined with each search request. Each of these will either help fine-tune the query or control the presentation of results.

Query Parsers

During query processing, Solr queries specific fields for matches to the user query. The fields may be a default set configured in advance or specifically defined in the query request. Each field has a type, and each field type has defined rules for how to index content of that type, and how to process queries of that content. In general, rules applied during indexing should be applied during queries to be confident of expected results. For example, if all fields are modified to lower-case during indexing, queries should be modified to lower-case to be sure they match as many terms as possible. These are defined in the field **analyzer** definitions, which include **tokenizers** and **filters** to be applied to indexing and queries. The tokenizers and filters will in many cases modify the original query from the user, perhaps by converting the user's input to lower-case or stripping extra characters like hyphens or other punctuation. There are several dozen options for tokenizers and filters and links at the end of this section will take you to more information about them. You can see the defined field analyzers by looking in the `schema.xml` file for the collection, or in the Admin UI [screens for Field Type](#).

While all of this may seem quite complicated, LucidWorks can be used out of the box with pre-set defaults. If the defaults do not match your desired behavior, however, learning a bit more about how Solr processes content during indexing and handles query requests may be required.

Related Topics

- [Apache Solr Reference Guide](#)
- [Tokenizers](#)
- [Filter Descriptions](#)
- [CharFilterFactories](#)
- [Language Analysis](#)


Constructing Solr Queries

In basic terms, searches are done with an HTTP GET that specifies the parameters to use for the search. As noted above, the `/lucid` request handler includes several components by default, which means they do not have to be added to the query. If using the `/select` request handler, however, items such as faceting and spell check suggestions would need to be specifically requested.

To search using the `/lucid` request handler, simply point your HTTP client or browser to <http://localhost:8888/solr/collection1/lucid?q=some+query>. LucidWorks returns XML by default. If you would rather have serialized PHP returned instead of XML, modify the URL to <http://localhost:8888/solr/collection1/lucid?q=some+query&wt=phps> and the response will be formatted in PHP.

Topics covered in this section:

- [Solr Query Parameters](#)
- [Query Parsers](#)
- [Related Topics](#)

 Any request sent to Solr must include the collection name. In the above example URLs, `collection1` refers to the default LucidWorks collection. If you have configured multiple collections, replace "collection1" with the appropriate collection name.

Solr Query Parameters

Solr has a tremendous amount of flexibility for controlling how queries are handled and how results are returned, all of which can be defined as parameters of the query. Some basic parameters to know, however are discussed below.

Parameter	Name	Uses	Example	Default
q	query	The main search request and keyword terms for the query.	q=solr	No s but t q.al as * find q.al defir none by tl

sort	sort	The field to sort the results by. Must also specify <code>asc</code> or <code>desc</code> to define the order. Multiple values can be used, separated by a comma. Multi-valued fields cannot be used for sorting.	sort=dateCreated+asc	score
fl	fields	The fields to return with the response.	fl=id,title	id, title, data, lastModified, mimeType, page
start	start	The number of results to skip when returning the results. Can be used with <code>rows</code> to provide pagination.	start=20	None. Lucid default is either 0 or 10.
rows	rows	The number of results to return. Can be used with <code>start</code> to provide pagination.	rows=15	None. Lucid default is either 10 or 20.
wt	writer	The response writer that Solr should use, which defines the format of the results.	wt=json	Solr's default is XML.
qt	query handler	The request handler to use to process the query. This can be used instead of a syntax like http://localhost:8888/solr/collection1/lucid? or http://localhost:8888/solr/collection1/select? shown in the examples above, or in conjunction with them to override the default request handler if one is defined.	wt=/lucid	/lucid default handler.

debug	debug	<p>Detailed information about the query and results, for debugging purposes. There are four options for this parameter:</p> <ul style="list-style-type: none"> • <code>true</code>: all of the debug information • <code>query</code>: information about the query only • <code>results</code>: information about the documents returned and how they scored • <code>timing</code>: information about how long each component took to complete their tasks 	debug=timing	In the Search Admin UI, the "expand" information (details, documents, scores)
-------	-------	--	--------------	---

There are many other parameters that can be employed, but these are the basic ones that let you submit a query and see some responses. For more detailed information on Solr's query capabilities (some of which depend on the query parser used), see the section of the Apache Solr Reference Guide on [Query Syntax and Parsing](#).



To ensure that your query is indexed and shown in the activity graphs in the LucidWorks Search Admin UI, include the `req_type=main` parameter in your query URL.

[Back to Top](#)

Query Parsers

All of query parsers included with Solr are available for use, in addition to the enhanced parser included with LucidWorks. This table shows what are considered the "main" query parsers that are designed for general use. There are also parsers that can be used for specific purposes, listed below.

Name	ID in LucidWorks	Description
Lucene or Solr	lucene	The Lucene Query Parser, with some Solr enhancements. In the Apache Solr Reference Guide, the section The Standard Query Parser has more details about the options for this parser.
DisMax	dismax	Search across multiple fields, allow +, -, and phrase queries while escaping most other Lucene syntax to avoid syntax errors. More information is available in the Apache Solr Reference Guide in the section The DisMax Query Parser .

Extended DisMax	edismax	A version of the Extended DisMax parser developed by LucidWorks and donated to the Apache Software Foundation for inclusion in Solr. More information is available in the Apache Solr Reference Guide in the section The Extended DisMax Query Parser .
Lucid	lucid	Allows Lucene syntax, enhanced proximity boosting, and query time synonym expansion. Tolerant of syntax errors. More information available in this guide in the section on the Lucid Query Parser.

There are also a number of query parsers which can be used on an ad hoc basis. Each of these are documented in full in the Apache Solr Reference Guide, in the section [Other Query Parsers](#). A few highlights include:

Name	Description
Boost	Generates a BoostedQuery which boosts a Query by a FunctionQuery.
Function	Parses a FunctionQuery which calculates a function over field values.
Field	Generates a query on a single field.
Nested	Delegates to another query parser, which can be used to override the default parser for a specific purpose.
Prefix Query Parser	Generates a prefix query on a single field.
Raw	Generates a raw unanalyzed term query.
Spatial Filter	Generates a query which filters results by a defined distance from a point in space.

Other query parsers are also available.

Related Topics

- [Query Syntax and Parsing](#), with several sub-pages for query parsers and local parameters

Solr Query Responses

- [Structure of the Response](#)
 - [The `responseHeader` Section](#)
 - [The `response` Section](#)
 - [The `highlighting` Section](#)
 - [The `facet_counts` Section](#)
 - [The `spellcheck` Section](#)
 - [The `debug` Section](#)
- [Format of Results](#)
- [Related Topics](#)

Structure of the Response

All Solr responses have at least two sections, the `responseHeader` and the `response`.

The `responseHeader` Section

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` Section

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.

The `highlighting` Section

The `highlighting` section will show, for each document in the response, the sections of text in the document that should be highlighted. If using the `/lucid` request handler, they will be shown as snippets of text, with HTML `` tags around them. Your client can consume those and you can format them by specifying the `highlight` class in your CSS however you'd like.

If using another request handler, such as `/select`, that does not have predefined configuration options for highlighting, you may need to set the parameters in your request. There are quite a few Solr parameters to control highlighting and the output in the response. For more details, see the section of the Apache Solr Reference Guide for [Highlighting](#).

The `facet_counts` Section

The `facet_counts` shows the facets that have been constructed for the result list, including the facet fields and facet values (with counts) to populate each field.

The `spellcheck` Section

The `spellcheck` will include suggestions for possible spelling errors in the user's query.

The `debug` Section

The `debug` section will contain the detailed information about how the query was processed. This section will only be returned if the `debug` parameter was used with the query.

There are many sub-section of this section, including:

- `explain`: Information about how each document scored according to the in relevancy ranking algorithm.
- `timing`: Information of how long each component took.
- `parsedquery`: The query string as submitted to the query parser.

Calculating the debug info, particularly the scores, is expensive in terms of processing power, so it should only be used when needed to debug query results.

Ack! What Do Those Scores Mean?

The `explain` sub-section of `debug` is the section that gives you information about the relevancy scores of each document returned in the query. It's the section you'll want to look at if you want to know why one document is ranked higher than another. But it's pretty complex.

The `explain` section shows you each factor that went into the final score and how it was weighted. There may be specific boosts defined (LucidWorks for example boosts a document when the query terms are found in the title, among others), the frequency of the term in the document may be high relative to the frequency of the term in all documents (a relationship called the "term frequency-inverse document frequency", or TF-IDF), or the term may have matched a field that is smaller than others (such as "author" instead of "body").

Some make the mistake of focusing on the score of a document in absolute terms instead of looking at a document's score relative to the other documents returned. This is an error because scoring of a single document is always relative to other documents in the index, and your index changes over time. The point of looking at scoring should be instead to understand why a document is ranked higher or lower than other document.

More information on `explain` can be found in the section describing the [Explain Info](#) of the LucidWorks Search UI.

[Back to Top](#)

Format of Results

The default format for search results in LucidWorks Search is XML. There are other options available - such as JSON, PHP, and CSV, among others - and you request the results in that format when sending the query. This is defined with the `wt` parameter.

The data is returned as a standard Solr search data structure, formatted either as XML, Ruby, Python, PHP, PHPS, and even server-side XSL. For more information, see the section in the Apache Solr Reference Guide on [Response Writers](#).

Related Topics

- [Understanding and Improving Relevance](#)
- [Explain Info](#)
- [Response Writers](#)

Query and Response Examples

LucidWorks Search includes a simple Search UI, but if you are going to build your own user interface, or your own application to access the data stored in LucidWorks, you will need to access the underlying engine directly.

LucidWorks is built on Apache Solr, so the techniques necessary for performing a search against it are the same as those for performing a search against Solr. In other words, an HTTP call to a URL of:

```
http://127.0.0.1:8888/solr/collection1/select/?q=NickChase
```

Would return a result such as this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">99</int>
    <lst name="params">
      <str name="q">NickChase</str>
    </lst>
  </lst>
  <result name="response" numFound="151" start="0">
    <doc>
      <str name="geo">none</str>
      <str name="id">29059644164939776</str>
      <int name="retweetCount">0</int>
      <str name="source">web</str>
      <str name="text">Working on a Twitter app; anybody got a preferred Java
Twitter library?</str>
      <arr name="text_medium">
        <str>NickChase</str>
        <str>en</str>
        <str/>
        <str>web</str>
        <str>Working on a Twitter app; anybody got a preferred Java Twitter
library?</str>
        <str>2011-01-23T06:15:33.000Z</str>
        <str>0</str>
      </arr>
      <date name="timestamp">2011-02-13T14:06:53.191Z</date>
      <arr name="userId">
        <str>999999999</str>
      </arr>
      <str name="userLang">en</str>
      <str name="userName">Nicholas Chase</str>
      <str name="userScreenName">NickChase</str>
    </doc>
    ...
  </result>
</response>
```

You can then consume that XML from within your application.

While XML is the default output format, LucidWorks supports multiple formats, including JSON, CSV, and even object formats such as PHP, Java, and Python.

In general, to change the output format, use the `wt` parameter, as in:

```
http://127.0.0.1:8888/solr/collection1/select/?q=NickChase&wt=json
```

This provides a response of

```
{
  "responseHeader":{
    "status":0,
    "QTime":1,
    "params":{
      "wt":"json",
      "q":"NickChase"
    }
  },
  "response":{
    "numFound":151,
    "start":0,
    "docs":[
      {
        "id":"29059644164939776",
        "userName":"Nicholas Chase",
        "userScreenName":"NickChase",
        "userLang":"en",
        "source":"web",
        "text":"Working on a Twitter app; anybody got a preferred Java Twitter
library?",
        "retweetCount":0,
        "timestamp":"2011-02-13T14:06:53.191Z",
        "geo":"none",
        "text_medium":["NickChase","en","","web","Working on a Twitter app;
anybody got a preferred Java Twitter library?",
        "2011-01-23T06:15:33.000Z","0"],
        "userId":["99999999"]
      }
      ...
    ]
  }
}
```

The structure of the results depends on the options you choose in the request string. For example, you can specify faceting and highlighting;

```
http://127.0.0.1:8888/solr/collection1/select/?q=twitter&facet=on&facet.field=userScreenNa
```

Which gives a result such as this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">359</int>
    <lst name="params">
      <str name="facet">on</str>
```



```

    <str name="facet.field">userScreenName</str>
    <str name="hl.fl">text</str>
    <str name="hl">true</str>
    <str name="q">twitter</str>
  </lst>
</lst>
<result name="response" numFound="2190" start="0">
  <doc>
    <str name="geo">none</str>
    <str name="id">38402455221829632</str>
    <arr name="objectType">
      <str>twStatus</str>
    </arr>
    <int name="retweetCount">0</int>
    <str name="source">&lt;a href="http://twitter.com/" rel="nofollow"&gt;Twitter
for iPhone&lt;/a&gt;</str>
    <str name="text">RT @Onventive: Really useful Twitter Android code RT @enbake
Developing an android twitter
      client using twitter4j http://is.gd/1YUFyY #a ...</str>
    <arr name="text_medium">
      <str>t4j_news</str>
      <str>en</str>
      <str/>
      <str>&lt;a href="http://twitter.com/" rel="nofollow"&gt;Twitter for
iPhone&lt;/a&gt;</str>
      <str>RT @Onventive: Really useful Twitter Android code RT @enbake
Developing an android twitter
      client using twitter4j http://is.gd/1YUFyY #a ...</str>
      <str>2011-02-18T01:00:33.000Z</str>
      <str>0</str>
    </arr>
    <date name="timestamp">2011-02-18T01:45:05.52Z</date>
    <arr name="userId">
      <str>88888888</str>
    </arr>
    <str name="userLang">en</str>
    <str name="userName">t4j_news</str>
    <str name="userScreenName">t4j_news</str>
  </doc>
  ...
</result>
<lst name="facet_counts">
  <lst name="facet_queries"/>
  <lst name="facet_fields">
    <lst name="userScreenName">
      <int name="beaker">189</int>
      <int name="cloudexpo">35</int>
      <int name="randybias">35</int>
      <int name="getjavajob">26</int>
      ...
    </lst>
  </lst>
</facet_counts>

```

```

    </lst>
  </lst>
  <lst name="facet_dates"/>
  <lst name="facet_ranges"/>
</lst>
<lst name="highlighting">
  <lst name="38402455221829632">
    <arr name="text">
      <str>RT @Onventive: Really useful <span
class="highlight">&#x201c;Twitter&lt;/span> Android code RT
      @enbake Developing an android <span
class="highlight">&#x201c;twitter&lt;/span> client</str>
    </arr>
  </lst>
  ...
</response>

```

Notice the structure of the search response: it starts with the `responseHeader` block, which provides information such as the query, whether you have specified highlighting, and so on.

Next is the `result` block, which shows the actual documents returned by the search, along with the `numFound` and `start` attributes, which specify the total number of results and the starting position for the results returned in this response. For each document, LucidWorks Search returns all fields that are marked as `stored=true` in the field definition.

If you have specified faceting, next you will see facet counts for each field specified. You can then use that information to build links to your narrowed search. For example, we started with the query:

```
http://127.0.0.1:8888/solr/collection1/select/?q=twitter&facet=on&facet.field=userScreenName
```

If you then wanted to build a link to results narrowed on the `userScreenName` `cloudExpo`, it would look like this:

```
http://127.0.0.1:8888/solr/collection1/select/?q=twitter&facet=on&hl=true&hl.fl=text&fq=userScreenName:cloudExpo
```

This way you have the same set of results, with the additional filter query of `userScreenName:cloudExpo`, which selects only the documents with a `userScreenName` field of `cloudExpo`.

After the facet information comes the `highlighting` block. Highlighting consists of snippets with the relevant information marked up appropriately. (By default, terms are marked up as a `span` with the class `highlight`, so you can use CSS to style them however you like.) Each snippet is contained in a block that refers back to the `id` value of the original document. So in this case, the `name` attribute of `38402455221829632` refers back to `doc` with an `id` of `38402455221829632`. You can then use this information to build your web application.

As far as how to actually use these responses, you can either work with them directly, or use the Solr API as provided for your programming language. For example, a SolrJ request looks something like this:

```
SolrServer server = new
CommonsHttpSolrServer("http://localhost:8888/solr/collection1");

SolrQuery query = new SolrQuery();
query.setQuery( "twitter" );
query.addSortField( "timestamp", SolrQuery.ORDER.desc );

QueryResponse rsp = server.query( query );
SolrDocumentList docs = rsp.getResults();
for (SolrDocument doc : docs){
    System.out.println((String)doc.getFieldValue("id")+": ");
    System.out.println((String)doc.getFieldValue("userScreenName")+ " --"
    +(String)doc.getFieldValue("text"));
}
```

Here you are creating a connection to the server, then creating and executing the request. From there, you can manipulate documents as you see fit.

APIs exist for most programming languages. You can find a list of bindings on the [Solr Wiki](#).

Related Topics

- [Searching chapter](#) from the Apache Solr Reference Guide

Understanding and Improving Relevance

Relevance is one of the most complex components of a search engine implementation, but it has a direct impact on how users perceive the value of the search system.

One of the reasons relevance is so complex is because two users performing the same query will likely have differing opinions about which documents best match their query. In the end, judging relevance has an inherent subjectivity to it. However, there are some ways to assess relevance and adjust how documents are scored to improve ranking. This section discusses the various approaches to analyzing a problem with relevance (real or perceived) and possible solutions.

For more background on how LucidWorks Search approaches relevance, see the discussion in the section on [Overview of Query Processing](#).

Topics in later sections:

- [Indexing and Relevance](#)
- [Queries and Relevance](#)
- [Relevance Tuning Tools](#)

Relevance Testing

Relevance should always be judged in the context of a specific index and a set of queries for that index. You should tune your relevance parameters for the types of queries users submit and the types of content you have indexed. For example, if you have an e-commerce site where users are accustomed to searching for your specific product names, and your content includes those names in the title, you might consider boosting title matches. If, however, your users do not know your specific product names very well, you might want to boost another field like color, or size.

When developing a search application, you will likely encounter issues with relevance during testing. Usually this happens when one or more users run their favorite query and aren't impressed with the results. This becomes a system bug that must be dealt with before launch. While the favorite-query approach can be useful, a more systematic approach may be more telling in the long run about how queries are and aren't being handled by the system.

An empirical approach uses real sample queries gathered from query log analysis. The top 50 or so queries are extracted from the logs, plus ten to twenty random queries. Next, one to three users enter each query into the system and then judge the top ten (or five) results. Judgments may be done on a scale of 1-5, with 1 being "relevant" and 5 being "embarrassing", or using another scale you determine. The goal of relevancy tuning is to maximize the number of relevant documents while minimizing the number of irrelevant ones. By recording these values and repeating the test over time, it becomes possible to see if relevancy is getting better or worse for the particular system in question.

An alternative method for judging relevance is to use what is commonly referred to as A/B testing. In this approach, some set of users are shown results using one version of the index while another set of users is shown the results from a different version. To judge the success of a particular approach, user clicks are tracked and analyzed to determine which approach provides better results.

Other approaches include log analysis on a beta site, letting users rate documents using a star (or similar) system, using third-party evaluation data sets such as TREC, or using focus groups. These approaches will all yield benefits, and you may want to adopt a combination of approaches, but empirical testing and A/B testing are the most comprehensive and give you easily repeatable results and verifiable results.

Once you have some data in hand about the scope of your problem, you are in a better position to understand what you want to try to improve and the changes you may need to make.

After Testing

Once you have identified that you want to make some changes to improve relevance of results, the next sections will discuss various approaches to doing so.

First, we cover some index-based approaches (things you do to documents as they are indexed), in the section [Indexing and Relevance](#).

Next, we cover query-based approaches (things you do to user queries), in the section [Queries and Relevance](#).

Finally, we'll cover [Relevance Tuning Tools](#).



Click Scoring Relevance Framework

One important aspect of LucidWorks relevance scoring functionality is the ability to boost documents that prior users have selected. This functionality is the [Click Scoring Relevance Framework](#) and can be enabled through the Administrative User Interface.

Related Topics

- [Relevance chapter](#) from the Apache Solr Reference Guide
- [Debugging Search Application Relevance Issues](#), by Grant Ingersoll, hosted at SearchHub.org.

Indexing and Relevance

For the most part, it is easier and more flexible to use query-time approaches to alter relevance ranking, but there are several techniques can be employed during indexing. These techniques almost always have to be mirrored on the query side, so they are only partially index-time approaches.

Stop words

Removing [stop words](#) (such as a, the, of, etc.) from the index and stripping them from queries is a common technique for reducing the size of an index and improving search results, despite the fact that it throws away information. While LucidWorks Search can remove stop words at [indexing time](#), it does not do so by default.

Removing stop words during indexing is now considered an archaic approach in most search applications. Instead, it is preferred to remove stop words from queries, except in certain types of queries where they are used to better clarify a user's intent (such as in phrases). Both the [Extended Dismax Query Parser](#) and the Lucid Query Parser can take advantage of stop words, see the section [Synonyms and Stop Words](#) for more information.

If stop words are removed from the index, you'll want to be sure to remove the same set of stop words from user queries. Not removing stop words at query-time when they have been removed from the index may actually reduce relevance by leading to a high number of unmatched terms from user queries.

Alternate Indexing Fields

When indexing, it is often useful to apply several different analysis techniques to the same content. For example, providing a default case-insensitive search is often the best choice for general users, but expert users will often want to do exact match searches which may additionally require a case-sensitive field. In Solr, this can be accomplished by using the `<copyField>` mechanism, as described in the Apache Solr Reference Guide section on [Copying Fields](#). In LucidWorks Search, this can be configured in the [Fields screen of the Admin UI](#), with the Fields API, or by editing the `schema.xml` file. If you use the Admin UI or the Fields API, you will not need to restart LucidWorks Search, but if you edit `schema.xml` by hand, a restart of [LucidWorks Search](#) will be required.

Other examples of times when alternate fields may be useful include applying different stemming approaches, using character-based and word-based n-grams, or stripping punctuation, accents and other marks. At query-time, you'll want to make sure to submit user queries to the fields that have had content analyzed the way you want.

Document and Field Boosting

When indexing using the Solr APIs it is possible to mark one document or field as being more important than other documents or fields by setting a boost value during indexing. These boost factors are then multiplied into the scoring weight during search, thus potentially boosting the result higher up in the result set. This type of boosting is usually done when knowledge about a document's importance is known beforehand. However, index time boosting only provides 255 distinct values of granularity and if a change is needed to the boost value, the document must be re-indexed.

In general, this type of index-time boosting is somewhat impractical: the field or document boosts must be included with the document every time the document is updated. If using one of the LucidWorks Search crawlers, this may be difficult to achieve without a workflow that includes [crawling as a batch](#), modifying documents offline, and then indexing the documents. In addition, the query-time boosting techniques offer much broader control over when and how boosts are applied.

However, LucidWorks Search also includes a way to boost fields in a document based on the length of the field. In theory, if a term that the user has searched for appears in a field that is significantly shorter than other fields (such as the title), it should be boosted more than if the term appears in a longer field (such as the body). The short field boost factor provides three approaches: "none", which provides no boost; "moderate", which uses the `LucidSimilarityFactory` to provide a smaller boost than the standard Lucene calculations; and "high", which uses Lucene's `DefaultSimilarityFactory` to calculate the boosts. This functionality is used during indexing - during query time, the standard Lucene calculations are used.

Stemming and Lemmatization

Stemming is the process of reducing a word to a base or root form. For example, removing plurals, gerunds ("ing" endings) or "ed" endings are all stemming techniques. Lemmatization is a variation of stemming that leaves a whole word in place, while stemming need not do that. There are many stemming theories and techniques. Some are quite aggressive, stripping words down to very small roots, while others (called light stemmers) are less aggressive.

LucidWorks includes many options for stemming but it is also possible to plug in a custom analyzer or use other Solr or Lucene analyzers not included. As a general rule of thumb, it is usually best to start with a light stemming approach that removes plurals and other basics techniques and then progress to more aggressive stemming only after performing some relevance testing as described in [Judging Relevance](#).

Default stemming in LucidWorks uses the Lucid Plural Stemmer for the default English text analysis Field Type which simply stems plural words into their singular form, although rules can be added to a rules file to protect and specially translate words or even add or modify stemming rules as needed (see the section [Lucid Plural Stemming Rules](#).) More aggressive stemmers are also available, like Dr. Martin Porter's [Snowball](#) stemmers (choose the "text (English Snowball)" Field Type).

To experiment with different stemmers, there is a well-defined mechanism in Solr for plugging in stemmers via the [Analysis Process](#). There is also an easy to use Admin interface for testing the analysis process located in the Solr Admin screens (access it via the "Advanced" tab of the Admin UI, or by going to <http://localhost:8888/solr/#/collection1>, replacing "localhost:8888" and "collection1" as needed for your environment).

Queries and Relevance

When working with queries to improve relevance ranking, there are a great number of tweaks and techniques that you can consider. In the section on [Relevance Tuning Tools](#), we'll discuss those smaller tweaks in more detail. But here we'll discuss some of the broader approaches you might consider.

One factor that shouldn't be overlooked is the importance of user education. While the techniques described below can make things much easier for users, educating users on how to use the proper query syntax, when to use it, and how to refine queries can be instrumental in enhancing the relevance of search results. Obviously, not all users will read manuals or take the time to learn new query syntax, so the following techniques can be used to achieve better results in many situations.

Boosting Specific Documents

The QueryElevationComponent in Solr provides a way to force specific documents to the top of the result list in response to a specific query. In Solr, it is configured with the `elevations.xml` file, but in LucidWorks Search it can be configured either with the [Search UI](#) or the Settings API.

This approach is useful if you have a few known documents that should always appear at the top for a query. It's also possible to force documents to not appear at all in the results for a query (i.e., "blacklisting") if that's required.

Query Term Boosting

Similar to [Document/Field boosting](#), terms in a query can be boosted. Boosting a query term implies that the term in question is somehow more important than the other terms in the query. One advantage of query time boosting is an expanded level of granularity is available for expressing the boost value. Additionally, the boost value is not "baked in" to the index, so it is easier to change.

You may also decide to give boosts if the user's term appears in specific fields, such as the title.

Click Scoring Relevance Framework

Available only in LucidWorks Search, this approach stores information about documents prior users have selected during their searches. The document ID and the user's query are recorded and then used to calculate boost values for those documents that are applied the next time the same query is submitted. Over time, the documents that have been clicked on the most will rise in the results list; if users stop clicking on the document, the algorithm has an aging factor that will cause them to gradually fall in the results list.

For more details, including how to enable Click Scoring, see the section [Click Scoring Relevance Framework](#)

Synonyms

Synonym expansion is a common technique that looks up each token in the original query and expands it with synonyms; strictly speaking, synonym expansion mostly improves the ability to get more documents (also called recall) rather than improving relevance ranking or excluding irrelevant documents. For instance, a user query containing "USA" could be expanded to "(USA OR "United States" OR "United States of America")", which may bring back results that the user intended to retrieve, but did not fully specify. If the user was looking for "USA" only, the results may be less relevant to him.

In LucidWorks Search, it is easy to specify a list of [synonyms](#) that can be used for expansion. Synonym lists are best created by analyzing query logs and then looking up synonyms for common query terms and then testing the results. Generic synonym lists (like those obtained from [WordNet](#)) can be useful, but care must be taken as too many synonyms can be problematic for users, especially if they are not appropriate for the genre of the index. It is, however, quite common to produce synonym lists contain common abbreviations, numbers (for example, 1 -> one, 2 -> two, and so on) and acronyms.

Unsupervised Feedback

Unsupervised feedback is a relevancy tuning technique that executes the user's query, takes the top five or ten documents from the result, extracts "important" terms from each of the documents and uses those terms to create a new query. The expanded query is executed and new results are returned to the user. This is all done automatically in the background with no interaction required by the end user. As an example, if the user searches for the word "dog" and the top three documents are (for the sake of example):

1. Great big brown dogs run through the woods.
2. Dogs don't like cats.
3. A poodle is a type of dog.

The feedback query might look something like (dog) OR (great OR big OR brown OR dog OR run OR woods OR cat OR poodle).

Since these terms co-occur with the word "dog" in high ranking documents, these terms may help further define a user's short query. Unsupervised feedback is often viewed as a helper, but it does rely on the assumption that the top few documents are highly relevant to the search. If they are not, then the results incorporating feedback will likely be worse than those without feedback.

Unsupervised feedback is optional in LucidWorks Search and is disabled by default. It may be enabled by checking the **Enable Unsupervised Feedback** check box in the [Querying Settings tab](#) of the Admin UI, or with the Settings API.

Supervised Feedback

Supervised feedback is similar to unsupervised feedback except that users explicitly pick which results are relevant, usually by clicking the result or checking a box indicating it is relevant. The LucidWorks Search feedback component does not currently support supervised feedback.

Boosting Documents According to Rules

You may have a complex suite of business rules (i.e., if user A is male, aged 25-35, display XYZ results first) that you'd like to apply. These may be built around profit or sales goals for the organization, but they may also be built around a deep knowledge of your users that you'd like to apply. In that case, you may need to integrate a Business Rules Engine. LucidWorks Search has provided an integration with Drools, but it's also possible to plug in other options. See the section [Business Rules Integration](#) for more details.

Related Topics

- [Options to Tune Documents' Relevance](#), by Tomàs Fernández Løbbe, hosted on SearchHub.org

Relevance Tuning Tools

Before starting to modify settings that impact how results are ranked, it's best to have an idea for the outcome you hope to achieve. Too often we have an emotional response to relevance, choosing a small number of favorite queries as our tests. However, as discussed in the opening section, you should run tests using queries that real users have submitted that have been pulled out of query logs. The scope of these tests is up to you and your available resources, but a methodical approach is preferred.

If you have done tests with real-world sample queries and had users (or internal testers) score results of those queries using a common scale, you have a way to quantify how "bad" the issue is before you make changes. This will allow you to quantify how much things improve for each proposed change, so you can base your decisions on data. This will also allow you to understand (and explain to stakeholders) some of the trade-offs you may need to make if your user's queries are improved but your CEO's favorite query is not.

If you do find you want to make changes, here are some tools and tips to assist you.

Relevancy Workbench

One way to experiment with system changes is to use the Relevancy Workbench, a new tool included with LucidWorks Search which allows side-by-side comparison of search results using different query parameters for two queries. This tool allows you to experiment with changes before making them permanently for all users.

Several parameters are available for experimentation, all of which relate to the fields that will be searched or the boosts that will be applied. A catch-all field is available for any parameters that aren't explicitly shown, making it a vital tool for testing the impact of any change you can think of.

The tool is available through the LucidWorks Search Admin UI, in the Relevance tab. See the [Relevance Help](#) for detailed information.

Explain Scoring

In the default LucidWorks Search UI, links will appear under each search result for "Explain"; clicking that will show the scoring of each document for the query. The scores cannot be tweaked here, but you can see the factors that make up the score and understand why the result appears where it does. This information can provide clues about why documents appear in the order that they do. The scores themselves are not the most important factor, but the scores of each document relative to other documents is telling.

More information about how to read explain scores is available in the section [Explain and Document Scoring](#).

Solr Analysis

Some problems may be deeper within the system, and may only be resolved by either changing how content is analyzed and transformed before indexing or changing how the user's query is analyzed and transformed. The field types defined for each field dictate this analysis and while LucidWorks Search includes sensible defaults, they are not universal and may need to be tweaked depending on your content.

The Solr Admin UI, which is available from the LucidWorks Search Admin UI through the Advanced tab, has a tool to help better visualize the analysis process which shows the outcome of each analysis step on both the indexing side and the query side. To use this tool, point a browser at <http://localhost:8888/solr/#/collection1/analysis> and enter the text to be analyzed. By trying out the text with different analysis capabilities (by selecting different Fields or Field Types), it is possible to better understand why matches may or may not occur.

More information about analyzers is available in the Apache Solr Reference Guide in the section [Understanding Analyzers, Tokenizers, and Filters](#).

Using Luke

Another useful tool for evaluating how documents have been indexed is [Luke](#), which is an easy to use GUI that provides valuable information about the underlying Lucene index. Its features include document browsing, query testing, term browsing (including high frequency terms) and statistics about the collection as a whole. To use Luke with LucidWorks Search, launch it using the script located in the `$LWS_HOME/app/luke` directory.

Once Luke is launched, point it at the LucidWorks Search index directory (such as `$LWS_HOME/data/solr/cores/collection1_0/data/index`, replacing "collection1_0" with the actual collection path you want to look at) and open the index. From there, the most useful actions are to view the high frequency terms, and also particular documents (under the Documents tab) using the "Browse by term" and "Browse by document number" options. Key items to look for are missing documents and fields, terms, or words that are not tokenized "correctly". Incorrect tokenization may not mean the analysis process was wrong, but rather the output is not what a user would expect.

Again, you probably wouldn't make changes with Luke, but it provides a deeper look into what is happening so you can make educated decisions about what should be changed, whether that is the analysis process for incoming content, the analysis process for user queries, or the default boost factors in play.



Luke in LucidWorks Search

LucidWorks Search packages a version of Luke, which is provided 'as is'. It can be found at `$LWS_HOME/app/luke` and launched by running the `luke.sh` script for Linux/Mac or the `luke.bat` script for Windows.

External Boost Data

The standard mechanism in Solr for adding external field data (which may affect ranking) is through the use of `ExternalFileField` type. This mechanism is sufficient when adding simple string or numeric values to be processed by function queries, but it's not sufficient to express more complex scoring mechanisms, based on other regular query types.

More information about external boost data is available in the Apache Solr Reference Guide in the section [Working with External Files and Processes](#).

Related Topics

- [Relevance tab](#) from the Help documentation for the Admin UI screen
- [Explain and Document Scoring](#) from the Help documentation
- [Luke](#)

Synonyms and Stop Words

Synonyms are words that are similar in meaning to each other, such as "hat" and "cap". In the context of a search application, they are another tool for improving results for users because they provide the opportunity to substitute words and expand the terms matched in the index.

Stop Words, on the other hand, are used to restrict the results of a search, by removing very small and very common words (such as "the" and "and") that often have little bearing on whether a document is a good match or not.

Synonym Expansion

LucidWorks Search manages synonyms with the use of a `synonyms.txt` file found in the `$LWS_HOME/conf/solr/cores/collection/conf` directory (unique for each collection). Synonyms can be edited in that file, via the Admin UI, or with the Settings API.

Synonyms can be either single terms or multi-term phrases. There are two ways to express synonyms:

- A comma-separated list of words (i.e., "lawyer, attorney" or "i-pod, i pod, ipod"). When the term entered by the user matches a term in the list, all terms are substituted for the term the user entered, including the matching term. If "lawyer, attorney" appears in the synonym list, when the user enters "lawyer", the system will search for documents that include both "lawyer" and "attorney".
- A mapping of one or more terms to another (i.e., "i-pod => ipod"). When entered as a mapping, the terms on the left of the "=>" symbol will be replaced by the terms on the right side of the symbol, which means that the user's query may not appear in the documents returned for the query. If "i-pod => ipod" appears in the synonym list, when the user enters "i-pod", the system will search for documents that contain the term "ipod" only.

There can be an unlimited number of terms and phrases which are defined as synonyms. However, it's usually not a good idea to add an entire thesaurus as a synonym file because not all terms are necessarily interchangeable (in some contexts, yes, but not always). For example, a doctor looking "myocardial infarction" is likely looking for documents that use the clinical term for the condition (and are thus more advanced) instead of documents written for a layman which likely uses the phrase "heart attack".

When considering synonyms, you should also consider which fields should be used for synonym expansion. In LucidWorks Search, the `body`, `description`, `title` and `text_all` fields are used for synonym expansion by default, meaning that those are the fields that will be used for the expanded or modified query.

If creating a synonym file manually, make sure to format the file properly. Lines starting with pound (#) are comments. Explicit mappings are indicated with terms separated by "=>", where a comma-separated list of terms on the left side will be replaced with the list of terms on the right side. Equivalent synonyms may be separated with commas and will give no explicit mapping (that is, the listed terms are equivalent). This allows the same synonym file to be used in different synonym handling strategies. For example:

```
lawyer, attorney
one, 1
two, 2
three, 3
ten, 10
hundred, 100
thousand, 1000
tv, television

#multiple synonym mapping entries are merged.
foo => foo bar
foo => baz
#is equivalent to
foo => foo bar, baz
```

If familiar with Solr, the file is formatted the same as the [Solr synonyms](#) file.

Stop Words

LucidWorks Search stores stop words in a file called `stopwords.txt`, found in the `$LWS_HOME/conf/solr/cores/collection/conf` directory (unique for each collection). The stop words can be edited in that file, via the Admin UI, or with the Settings API.

The stop word file is just a list of terms, one per line.

Many common prepositions, pronouns, and adjectives offer little benefit for matching documents, but can add some value when ranking results. Although it is possible to remove stop words [when documents are indexed](#), more relevant results will be achieved by indexing all terms, querying only non-stop words, and then boosting the results by including the stop words with non-stop words. There is the special case where a query consists only of stop words (such as the classic, "To be or not to be"). In that case, all words are included in the query.

All words within quoted phrases are used for the query, even if they are stop words. The user can also force a stop word to be included in the search by either preceding it with a plus sign ("+") or enclosing it within double quotation marks. For example,

User Input	Query Interpretation
at a conference	"at" and "a" are stop words, so they will not be included with the query

+at a conference	"at" will be included in the query, but "a" will not
"at" a conference	Same
"at a conference"	All three words will participate in the query
this is it	There is no need to override because all three words are stop words, so all three will be included in the query

If creating the stop words file manually, the format is one term per line, as in:

```
a
an
and
are
as
at
```

This is the same format as the [Solr stopwords format](#).

Related Topics

- [Suppressing Stop Word Indexing](#)
- [Settings API](#)
- [Synonyms in the Admin UI](#)
- [Stop words in the Admin UI](#)

Suppressing Stop Word Indexing

This functionality is
not available with
LucidWorks Search
on AWS or Azure

By default, LucidWorks Search indexes all [stop words](#). Modern data storage is very cheap and even the simplest of stop words provide additional context that boosts relevancy and enables more precise queries. By default, the Lucid query parser eliminates stop words from basic queries, including them only when they are used in quoted phrases, or when the query term list consists only of stop words. In addition, the Lucid query parser uses query stop words to construct relevancy boosting phrase terms (bigram and trigram phrases) to supplement the basic query. Still, there may be applications and environments where the choice is to suppress the indexing of stop words.

TODO - update this for Field Types in the UI and API

Disabling Stop Word Indexing

Solr field types in the schema XML file control whether stop words will be indexed for particular fields. A stop word filter may be placed in the tokenizer chain for the index analyzer for a field type to filter out stop words and assure that they will not be stored in the index.

Filters are specified at the field *type* level, not the field level. For example, you may have `title` and `body` fields, both with the `text_en` field type. A stop word filter may be specified for the `text_en` field type and will apply to all fields of that same type, in this case `title` and `body`. If you really need to have a separate filter for a subset of the fields of a given type, you must create a separate field type to use for that subset of fields.

The standard stop word filter is named `StopFilter` and is generated by the `StopFilterFactory` Java class. LucidWorks ships with a schema XML file (`schema.xml`) with the `text_en` field type with a commented out entry for this standard stop word filter. To enable it, simply remove the XML comment markers around that one filter entry.

Schemas are Collection Specific

The `schema.xml` file is specific to each collection and can be found under `$LWS_HOME/conf/solr/cores/collection/conf`. If using multiple collections, be sure to locate the correct `schema.xml` file for the collection to be updated. After editing the `schema.xml` file, LucidWorks should be [restarted](#). On some Windows machines, LucidWorks may need to be stopped before editing the file.

So, starting with the following in `schema.xml`:

```
<fieldType class="solr.TextField" name="text_en" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <!-- 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.StopFilterFactory" ignoreCase="true"
           words="stopwords.txt"/>

    -->
    <filter class="solr.WordDelimiterFilterFactory"
           generateNumberParts="1" generateWordParts="1"
           catenateAll="0" catenateNumbers="1" catenateWords="1"
           splitOnCaseChange="0"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ISOLatin1AccentFilterFactory"/>
    <filter class="com.lucid.analysis.LucidPluralStemFilterFactory"
           rules="LucidStemRules_en.txt"/>
  </analyzer>
  ...
```

Edit the stop filter factory entry that is commented out:

```
<!--
<filter class="solr.StopFilterFactory" ignoreCase="true"
       words="stopwords.txt"/>

-->
```

And remove the XML comment markers to get:

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

Which results in the following analyzer description:

```
<fieldType class="solr.TextField" name="text_en" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
    <!-- 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.StopFilterFactory" ignoreCase="true"
          words="stopwords.txt"/>
    <filter class="solr.WordDelimiterFilterFactory"
          generateNumberParts="1" generateWordParts="1"
          catenateAll="0" catenateNumbers="1" catenateWords="1"
          splitOnCaseChange="0"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.ISOLatin1AccentFilterFactory"/>
    <filter class="com.lucid.analysis.LucidPluralStemFilterFactory"
          rules="LucidStemRules_en.txt"/>
  </analyzer>
  ...
```

After such a change, be sure to [re-index all documents](#).

Also, make sure that the query analyzer for that field type references the same stop words file:

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

Do not change or comment out the query analyzer when making this index change.



This example only changes the `text_en` field type. If other field types are being used, or should be changed, find the section of the `schema.xml` for that field type and

Position Increment Mode

There are two modes for suppressing stop word indexing:

1. **Skip mode:** Completely ignore or skip them, as if they were not present. This is the default when no other option is selected. When skip mode is selected, the query parser will ignore or skip stop words in quoted phrases.
2. **Position increment mode:** Do not store them in the index, but increment the position counter so as to leave a blank at the position of each stop word. When position increment mode is selected, the query parser will also skip each stop word, but will increment the position of the next term in the phrase so as to allow any term to match between the previous term and the next term after the stop word. This will allow for more precise query matching than the first mode where stop words are simply discarded.

For example, given these documents:

- Doc #1: Buy the time for the test.
- Doc #2: Buy more time for the test.
- Doc #3: Buy time for test.

A query of `Buy the time` regardless of the stop word indexing mode will be equivalent to `Buy AND time` and match all three documents.

A query of `"buy the time"` in normal indexing mode will match exactly that phrase and match only the first document. In skip mode it is equivalent to `"buy time"` and will match the first and third documents. In position increment mode the query is equivalent to `"buy * time"` which is not a valid query format but indicates that `"time"` will match the second word after `"buy"` regardless of the intervening word. This will match the first and second documents, but not the third document.

To enable position increment mode, edit the `StopFilterFactory` entry of the index analyzer (which was un-commented above) in `schema.xml` to add `enablePositionIncrements="true"`. The section will appear as follows:

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

Only the index analyzer should be changed. The query analyzer should not be changed regardless of the indexing mode. The query parser has internal logic that decides whether and when to call the query stop word filter.

After this change, be sure to [re-index all documents](#).

Spell Check

Spell check, also known as "Did You Mean?", is the ability of the search application to make alternate suggestions for queries based on words that are similar to the terms entered by the user.

Integrated query spell checking is bundled with LucidWorks Search, with the option to integrate third-party enhanced spell checking capabilities. It is index-driven, meaning all suggestions are derived from the actual content in an indexed collection and not from a predefined dictionary of words. In practical terms, this helps solve problems with messy data written by a variety of authors of varying quality where one author may spell a word one way, while another author spells it a different way and the user spells it a third way. An index-derived spell checker provides suggestions based on the (sometimes incorrect) words in the dictionary, ensuring that end users still find relevant documents even if they contain misspellings.

To enable spell checking for specific fields, three steps must be taken:

1. Enable spell checking by accessing the [Querying - Settings](#) tab of the Admin UI and check the box next to "Spell-check". Alternatively, the Settings API can be used.
2. Ensure there are fields configured for spell checking by accessing the [Indexing - Fields](#) tab and choosing "Index for Spell Checking". The Fields API could also be used to modify field settings. Be sure to select fields that contain ample text-based content that end users are going to search against using word-based queries. For example, the title and body fields are good candidates, while a "price" field likely isn't.
3. Crawl your content.
4. Perform queries.



Spell Check Settings are Per Collection

The indexes created for spell checking are unique to each collection, and based on the documents indexed for a particular collection. In a multi-collection environment, the steps to enable spell checking must be done in each collection.

When indexing content, LucidWorks will automatically create an index of terms to be used for term suggestions. By default, LucidWorks will create this index from content in the `author`, `body`, `description`, and `title` fields.



In prior versions of LucidWorks, a separate task needed to be scheduled to build the spell check index of terms. Starting with v2.0 of LucidWorks Search, that requirement has been removed and the `spell` index will be created automatically during regular indexing.

Related Topics

- [Query Settings](#)

-
- Settings

Auto-Complete of User Queries

Query auto-complete shows users suggestions for their queries as they type the words. In LucidWorks Search, this is an index-driven feature, meaning all suggestions are derived from the actual content in an indexed collection and not from a predefined dictionary of words. For users, this means they will see suggestions for actual terms in documents, not for terms that may not exist in the content.

Auto-Complete Settings are Per Collection

The indexes created for auto-complete are unique to each collection, and based on the documents indexed for a particular collection. In a multi-collection environment, the steps to enable auto-complete must be done in each collection.

To enable auto-complete of user queries, three steps must be taken:

1. Enable auto-complete by accessing the [Query Settings](#) screen of the Admin UI and check the box next to "Auto complete". Alternatively, the Settings API can be used.
2. Ensure there are fields configured for auto-complete by accessing the [Indexing Fields](#) screen and choosing "Index for autocomplete". The Fields API can be used instead if you prefer. A good auto-complete field is a field that contains ample text-based content that end users are going to search against using word-based queries. For example, the title and body fields are good candidates, while a "price" field probably isn't.
3. After crawling some content, create the "autocomplete" index by accessing the [Index Settings](#) page and scheduling a time for the "Generate autocomplete index" job to run. The Activities API can be used instead if preferred. This **must** be done before automatic query completion will occur for users.

LucidWorks Search does not create the auto-complete index by default. Auto-Complete indexing jobs must be scheduled using the [Indexing - Settings tab](#) of the Admin UI (or via the Activities API) before query suggestions will appear for users.

- ✔ If you enable auto-complete but don't see any suggestions, you may want to modify the `threshold` parameter, which defines the minimum fraction of documents a term should appear in before being added to the `autocomplete` index. The default is "0.05" (or 5%), and a lower number will include more terms in the index. A smaller number may be helpful when just starting out with a small sample set of documents.

To modify this parameter, edit `solrconfig.xml` for each collection (in `$LWS_HOME/conf/solr/cores/collection/conf`). Find the section:

```
<searchComponent class="solr.SpellCheckComponent" name="autocomplete">
```

Find the parameter `<float name="threshold">.005</float>` and change it to the desired value. After saving `solrconfig.xml`, [restart LucidWorks](#).

Automatic Creation of Auto-Complete Indexes

This functionality is
not available with
LucidWorks Search
on AWS or Azure

By default, LucidWorks does not build the indexes for auto-complete each time documents are added to the index because doing so may have performance implications in a production environment with a large index. However, LucidWorks can be configured to do this automatically by changing the `buildOnCommit` setting in `solrconfig.xml` to `true`. Usually, it's a better idea to schedule index builds so that they run on a regular interval rather than doing it on every commit using this method.

If, however, you would like this to happen automatically, find the following section in the `solrconfig.xml` file for each collection:

```
<!-- Auto-Complete component -->
<searchComponent name="autocomplete" class="solr.SpellCheckComponent">
  <lst name="spellchecker">
    <str name="name">autocomplete</str>
    <str name="classname">org.apache.solr.spelling.suggest.Suggester</str>
    <str name="lookupImpl">org.apache.solr.spelling.suggest.tst.TSTLookup</str>
    <str name="field">autocomplete</str>
    <str name="storeDir">autocomplete</str>
    <str name="buildOnCommit">false</str>
    <float name="threshold">.005</float>
    <!-- <str name="sourceLocation">american-english</str> -->
  </lst>
</searchComponent>
```

In the section, `str name="buildOnCommit">false</str>`, change "false" to "true", and save the file. [Restart LucidWorks](#) for the changes to take effect. Repeat this for each collection that should build the auto-complete index each time documents are added to the index.

Enterprise Alerts

The alerts feature of LucidWorks Search allows a user to save a search and receive notifications when new results are available.

A *passive alert* acts like a smart saved search. It is smart in the sense that it keeps track of the last time the user checked for new results to their search and provides only new results the next time the alert is checked. As the name implies, a passive alert provides no notification when new documents are indexed. It waits for a request before it checks for new query results.

An *active alert* is checked periodically at a user-defined interval (currently every hour, day or week is available). When new results to the query are discovered, an active alert sends a notification via email to the email address defined in the alert. At the current time, only email notifications are possible.

How Alerts Work

1. The user does a search, and clicks the link under the search box to "Create new alert".
 1. The user configures the alert and notification settings, including how often to run the alert (`period`) and an email address to send alert notifications.
 2. LucidWorks Search automatically saves the timestamp of when the alert was created (`checked_at`).
2. Every 60 seconds, a scheduled process within the UI checks to see if it is time to run any alerts.
3. When the alert is run, the query is executed as entered by the user, on the collection that the query was initially run on, and the timestamp of the most recent document is compared to the timestamps of documents in the result set.
4. If there are new results for the user, a notification is sent, assuming the mail server has been configured in the [Settings](#) page of the UI.

Parameter names in parentheses above refer to the attributes used with the Alerts API. Alerts can be set up with the default Search UI, but while designing your own search application, you will likely need to use the Alerts API to integrate the functionality.

Enabling Alerts

In order for alert notifications to work with LucidWorks Search, the email server must be configured via the [System Settings](#) page in the Admin UI.

In addition, the LucidWorks Search schema must define a `timestamp` date field. Both active and passive alerts require that the index define a `timestamp` date field that is indexed, defaulted to NOW, and used to indicate the time of document indexing. By default, LucidWorks Search schema already defines this field. However, if modifying the LucidWorks Search default field set (the "schema"), you must retain this field for alerts to work properly.

Click Scoring Relevance Framework

One way to modify how results are ranked for users is to adjust the scoring of results based on user feedback (either explicitly or implicitly). Query logs provide a wealth of information that indicates what users were searching for and which results they found relevant to the query. If certain documents are often selected as answers to queries, it makes sense to increase their ranking based on their popularity with users.

LucidWorks Search includes a component that enables administrators to add this type of information to the index. This component is referred to as the Click Scoring Relevance Framework (or Click Scoring, for short). The framework includes tools for query log collection, log processing, and robust calculation of log data to boost certain documents. It is possible to supply boost data prepared without Click Scoring tools included with LucidWorks, however the data must be available in a predefined location and follow a specified text format. More details about how Click Scoring works and information about advanced configuration parameters are described in [Using Click Scoring Tools](#).

This component can be enabled in the [Query Settings](#) section of the Admin UI or with the `click`-related parameters of the Settings API. Once enabled, a job must be scheduled to process the click logs and create the data for boosting documents based on prior clicks.



There is currently a known issue where Click Scoring will not properly process calculated boost information until LucidWorks Search is restarted. So, when enabling Click Scoring, please also schedule a full LucidWorks Search restart. For details on how to restart, see the section [Starting and Stopping LucidWorks Search](#).

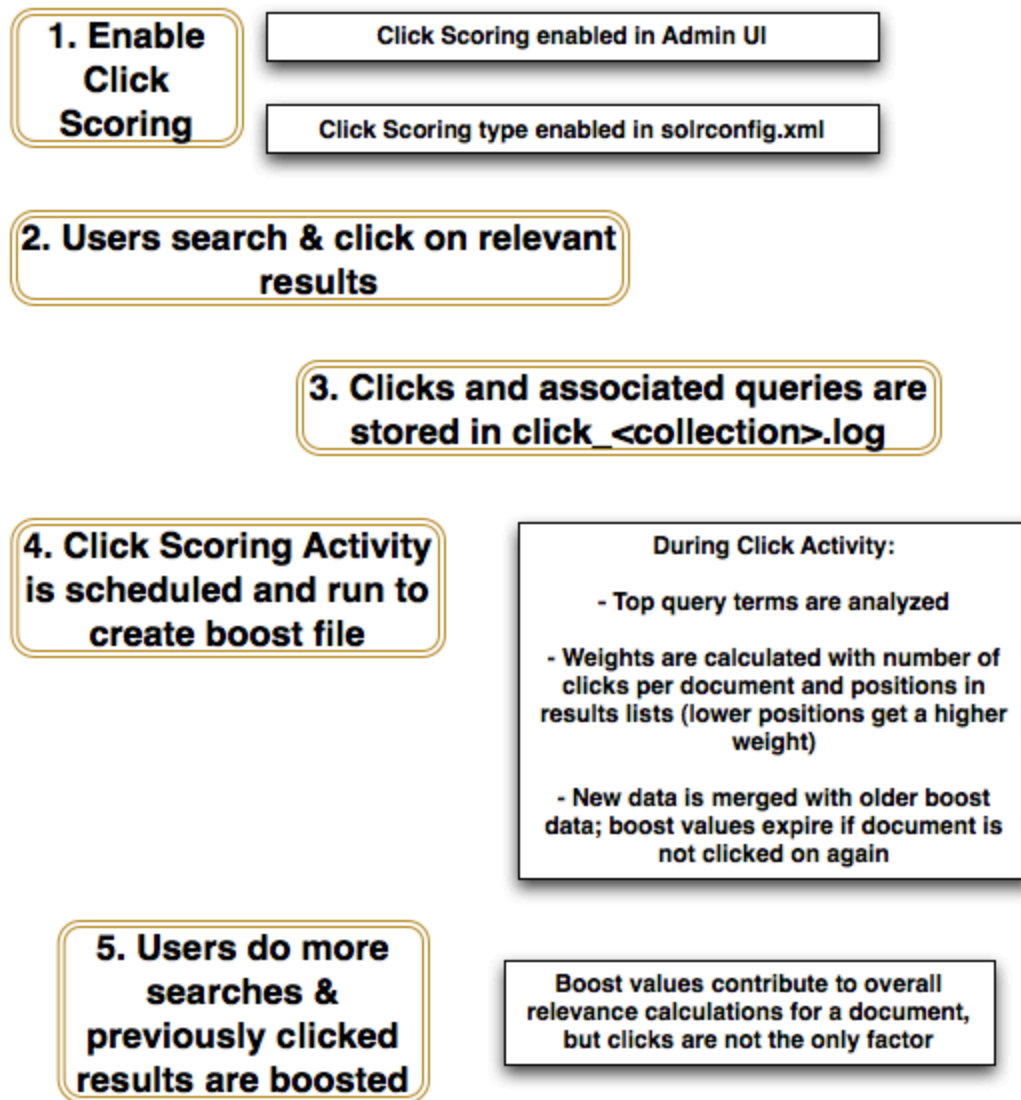
Topics covered in this section:

- [Functionality of Click Scoring](#)
 - [Collection of Query Terms and User Clicks](#)
 - [Processing Logs](#)
 - [Maintenance of Historical Click Data](#)
 - [Document Boost Data](#)
 - [Integration of Boost Data with the Index](#)
- [Using Click Scoring information](#)
- [Related Topics](#)

Functionality of Click Scoring

When users select a particular document for viewing from a list of search results, we can interpret this as implicit feedback that the document is more relevant to the query than other documents in the results list. We can infer a strong association between the terms of the query and the selected document, because users have shown through clicks that they believe the selected document matches their query better than other returned documents.

This graphic gives an overview of how Click Scoring works:



The reinforcement of ranking and terms is counterbalanced by the "expiration" of the past history of click-through events, to avoid situations when documents that are selected many times start to permanently dominate the list of results. Without expiration of old history, these results may become selected even more often at the expense of other perhaps more relevant documents that did not enjoy such popularity over time.

Click Scoring implements several major areas of functionality related to the processing of click-through events:

- collection of query logs and click-through logs
- maintenance of historical click data to control the expiration of past click-through events
- aggregation of log data, calculation of click-induced weights and association of query terms with target documents
- integration of boost data with the main index data

These areas of functionality are described in the following sections.

Collection of Query Terms and User Clicks

Records of user clicks include two pieces of information: the document ID and the query term entered by the user.

The default LucidWorks Search UI records user clicks automatically when Click Scoring has been enabled. When you write your own search application, you will need to make calls to the Click Scoring API to record user clicks and query events.

Both the queries and the user clicks are logged to the same log file. The default location of this file is in `$LWS_HOME/data/logs/click-collection.log`, where `collection` is the name of the collection (for example, `click-collection1.log` contains clicks to the the default LucidWorks collection, `collection1`).

When using [Index Replication](#) this log data is not replicated to slave nodes. Since the Click Scoring API points to the LucidWorks Search Core component, which is only used on a single node, and not directly to the indexes, it is not required to replicate the log files across shards. The latest version of the calculated boost data (after the logs have been processed) is replicated together with the main index files, this allows the slave nodes to perform click-based scoring in the same way as the master node that calculated the boost data.



Click Scoring is not available in SolrCloud mode.

Processing Logs

Whether generated by the default LucidWorks Search UI or from your own application with the Click Scoring API, the Click Scoring log files must be processed to calculate boost values. This processing step can be started with the Activities API, or scheduled to run periodically using the Admin UI by setting a recurring activity in the [Index Settings](#) screen of the Admin UI.

This process results in the creation of calculated click boost data, which is by default located in `$LWS_HOME/data/solr/cores/collection/data/click-data/current`.

Maintenance of Historical Click Data

Each time the Click Scoring logs are processed, the system stores a copy of the current `click-collection.log` (by default, this is in `$LWS_HOME/data/solr/cores/collection/data/click-data/`). Other data produced during Click processing is also stored in that location.

Over time the amount of data collected could be significant. LucidWorks Search does not delete this data automatically, because query and click-through logs are a valuable resource and can be used for other data mining tasks. If the size of this data becomes a concern, all subdirectories in that location can be removed except for `current/` and `previous/` directories that preserve the current and previous boost data.

Document Boost Data

The final boost data file follows a simple text format, so the boost data can be also supplied by an external process if desired. See [Using Click Scoring Tools](#) for more details about the structure of the boost data file.

Integration of Boost Data with the Index

If Click Scoring is enabled and logs have been processed, the boost data is integrated on the fly with the main index when new documents are indexed, an index optimization is run, or a full re-index is executed. Most frequent query terms are added as a field to respective documents, and weights of these documents are adjusted.

The field names added by Click Scoring are configurable, but assuming their prefix is set to the default value of `click` the following fields will be created from boost data and automatically populated:

- `click`: an indexed, not-stored field with a single term "1", whose purpose is only to convey a floating-point field boost value. Field boost values have limited resolution, which means that small differences in boost value may yield the same value after rounding.
- `click_terms`: an indexed, stored, and analyzed field that contains a list of top terms associated with the document (presumably obtained through analysis of click-through data). This field's Lucene boost is also set to the boost value for this document obtained from the boost data file.
- `click_val`: an indexed, stored field that contains a single term: a string representation of the boost value for this document. This format is suitable for processing in function queries.



Using Click Scoring with NearRealTime Search

Enabling Solr's Near RealTime (NRT) search by configuring the `update_handler_autosoftcommit_*` parameters with the Settings API or the Auto-soft-commit* settings in the [Admin UI](#) has some impacts on how user clicks are processed by LucidWorks.

In order to avoid performance issues with NRT search when Click Scoring is enabled, documents added between the last "hard" commit and the current "soft" commit are **not** augmented with click-through data.

Deletions since the last hard commit are processed as usual (i.e., documents deleted are not visible), but their term statistics are still included in the global term statistics (which includes the fields added by Click). Added documents since the last hard commit will not get any click-related fields until the next hard commit, even if a document with the same unique key was deleted and replaced by a new, updated, version of the document.

Using Click Scoring information

There are several ways that Click Scoring information can affect ranking of results. By default, LucidWorks Search is configured to use Click Scoring data as an additional field in a query parsed by the Lucid Query Parser. [Other methods](#) can be configured manually, and may involve using `click_val` field as an input to a function query. This section describes the `lucid` query parser method, which is the default.

When [Click Scoring](#) is enabled via the Admin UI, a boost field `click_terms^5.0` is automatically added to the list of fields for the search handler (which uses a `lucid` query parser). This means that query terms will be matched with the `click_terms` field using the relative weight of 5.0. This weight can be changed with the Settings API or by editing `solrconfig.xml` if you'd like a larger or smaller boost.

The end result of this query processing is that documents that contain in their `click_terms` field terms from the query will have a higher ranking, proportionally higher to the popularity of the document (the number of click-throughs) and the overlap of query terms with `click_terms`. It may be difficult, however, to see the effects of integrating Click Scoring boosts from only a few clicks on a document during testing. This is because the actual boost that occurs The score contribution of this match will be related to this weight, the term frequency/inverse document frequency scoring formula for this field, and the usual `lucid` (extended `dismax`) scoring rules.

Related Topics

- [Using Click Scoring Tools](#)

Using Click Scoring Tools

This functionality is
not available with
LucidWorks Search
on AWS or Azure

The Click Scoring Tools package is a set of tools for analyzing query and click-through logs in order to obtain relevance-boosting data. This boost data can then be used by other Click Scoring components such as `ClickIndexReaderFactory` and the `lucid` query parser to adjust document ranking based on the click-through rate and common query terms.

File Formats

The Click Scoring Tools package reads and generates files that follow specific formats, which are summarized below. All files are plain text files with tab-separated records, one record per line.

Query and Click-through Log Format

Click Scoring tools expect this file to be located in
`$LWS_HOME/data/logs/click-<collectionName>.log`.

```
Q TAB queryTimestamp TAB query TAB requestID TAB numberOfHits
C TAB clickTimestamp TAB requestID TAB documentID TAB position
```

The fields are:

Field	Description
Q or C	Identifies the type of the record, either a query log record or a click-through log record
queryTimestamp	A long integer representing the time when the query was executed
query	The user query, after basic escaping (removal of TAB and new-line characters)
requestID	A unique request identifier related to query and timestamp
numberOfHits	The total number of results matching the query
clickTimestamp	A long integer representing the time of the click-through event
requestID	The same value as above for the Q record
documentID	The <code>uniqueKey</code> of the document that was selected
position	The 0-based position of the selected document on the list of results

Boost File Format

This file is usually generated as a result of the Click Scoring processing of log files, but it could be also supplied by some other external process. Click Scoring expects this file to be located in `$LWS_HOME/data/solr/cores/collection/data/click-data/current`.

```
documentID TAB list(topTerms) TAB list(boost) TAB list(updateTimestamp)
```

The fields are:

Field	Description
documentID	The uniqueKey of the document
list(topTerms)	A comma-separated list of pairs in the format phrase:weight
list(updateTimestamp)	A comma-separated list of long integer timestamps, which affect how the current boost data will be aggregated with the next version of boost data. This element is optional and it's for internal use by Click Scoring Tools

Click-induced Boost Calculation

When Click Scoring tools are run (using the [ClickAnalysisRequestHandler](#)) old boost data (if present) is merged with the new boost data, processed by a `BoostProcessor` to produce the new numeric boost value per documentID, and a new list of top-N shingles per documentID. Previous values of the floating-point boost are preserved in a boost history field, so that they may be considered during the next round of calculations.

The default configuration uses a `BoostProcessor` that discounts historical boost values depending on the passed time by applying an exponential half-life decay formula. Such discounted historical values are then aggregated with the current values. This method of aggregation reflects both past history of click-throughs and also reacts closely to recent click-through events.

ClickAnalysisRequestHandler

The `ClickAnalysisRequestHandler` initiates and monitors the click-through analysis. The tools for Click Scoring processing are available via `com.lucid.handler.ClickAnalysisRequestHandler`, which can be activated from the `solrconfig.xml` configuration file the same way as any other request handler.

The configuration that ships with LucidWorks Search already contains a section that activates this handler, under the relative path `/click`.

This handler accepts a `request` parameter, which can take one of the following values:

STATUS: return the status of the ongoing analysis, if any. Example request:

```
curl http://localhost:8888/solr/collection1/click?request=STATUS
```

Example response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">205</int>
  </lst>
  <str name="logDir">java.io.File:.../logs</str>
  <str name="prepDir">java.io.File:.../click-prepare</str>
  <str name="boostDir">java.io.File:.../click-data</str>
  <null name="dictDir"/>
  <str name="processing">Idle.</str>
</response>
```

PROCESS: start the clickthrough processing. If the processing is already running, an error message will be returned and this request will be ignored.

Example request:

```
curl http://localhost:8888/solr/collection1/click?request=PROCESS
```

Example response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">136</int>
  </lst>
  <str name="result">Clickthrough analysis started.</str>
</response>
```

Subsequently, the status returned after all processing is finished will look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <str name="logDir">java.io.File:./logs</str>
  <str name="prepDir">java.io.File:./click-prepare</str>
  <str name="boostDir">java.io.File:./click-data</str>
  <null name="dictDir"/>
  <str name="processing">Stopped: Stage 3/3: prepare=finished, ok aggregate=finished, ok
  boost_calc=finished, ok</str>
</response>
```

STOP: stop the currently ongoing analysis, if any is running.

Example request:

```
curl "http://localhost:8888/solr/collection1/click?request=STOP"
```

Example response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">0</int>
  </lst>
  <str name="result">There is no running analysis to stop - ignored.</str>
</response>
```

When processing is finished, new versions of boost files will be placed in the `current` directory, and previous boost data will be moved to the `previous` directory. At this point in order to read the new boost values SolrCore needs to be reloaded (for example, by issuing a `<commit/>` update request).

In addition to the `request` parameter this handler supports also the following parameters:

- `commit` (default to false) if set to true, then after the processing is finished the handler will automatically execute a commit operation to reopen the IndexReader and to load the newly calculated boost data. Please note that Solr supports only a single global commit, which means that all other open transactions (such as ongoing indexing) will also be committed at this time.
- `sync` (default to false) if set to true, then the processing will be executed synchronously, blocking the caller and returning only when all processing is finished. Default is to run the processing in a separate background thread.

Click Scoring Tools and Index Replication

When LucidWorks Search is configured to use [Index Replication](#) the boost data files (by default, in `$LWS_HOME/data/solr/cores/collection/data/click-data`) will also be automatically replicated. Due to the internal limitations of Solr's `ReplicationHandler` the boost data file will be located *inside* the main index directory on the slave nodes, but it will be properly recognized by the Click Scoring components on the slave nodes.

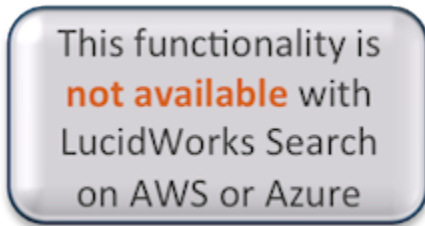


Click Scoring does not currently work with the [SolrCloud](#) functionality available with Solr 4.

For the replication of `boost.data` to work the `solconfig.xml` must contain the following line in the `<mainIndex>` section:

```
<mainIndex>
  ...
  <deletionPolicy class="com.lucid.solr.click.ClickDeletionPolicy"/>
  ...
</mainIndex>
```

Business Rules Integration



LucidWorks Search integrates the Business Rules Module available for Solr installations in the LucidWorks Marketplace. In v2.6.3, this replaces the prior implementation of business rules.

About Rules Engines

A *rules engine* is designed to allow business users to write rules that effect the processing of search results. For instance, an e-commerce company may wish to alter the search results to boost particular documents based on a sale, or the HR department of a company may wish to make sure the document covering 401K benefits is always at the top of a search for 401K. In essence, a rules engine integrated with a search engine allows businesses to dynamically impact relevance of results based on business needs without having to write extensive, low-level client-server code. Instead, they can express rules in a declarative programming language that are much simpler to understand without the complexity of logic that goes into writing code in a programming language like Java or Ruby.

All business rules depend on information from the system to analyze and take actions. This information is known to the rule processor as *facts* which will be present in the *knowledge session*. LucidWorks Search will add facts to the knowledge session on each request and the user's business rules can use and manipulate those facts.

In a rules engine, users express rules to be matched along with instructions in case a rule is matched, using simple if-then statements. The rules engine then figures out which rules should be fired given the facts present in the system. For example, a set of rules may look like:

```
if owner.hasDog then recommend dog food
if owner.hasCat then recommend cat food
if owner.gender is female and store is "sporting goods" then discount golf clubs 20%
```

The important thing to note in this example is we didn't have to do any complex logic to tie these rules together. We simply express the conditions and the things that should happen if a condition is true. The engine is responsible for figuring out which rules should fire based on the information (facts) it has to work with when evaluating the rules. It is also important to note that at any given execution of the engine some, all, or none of the conditions may be met depending on the facts in the system, thus implying that all of the "then" clauses will be executed.

When Should I Use Business Rules?

There is a time and place for the use of business rules. Generally speaking, they are most effectively used in situations where non-developers are expected to apply changes to the search results based on business conditions. They are not a replacement for code that integrates search into an application, but instead should be thought of as a way for companies to fine tune user interactions with a system without the need to go through extensive (and expensive) development cycles. It also is not a substitute for general relevance tuning across a broad set of queries nor is it appropriate for ranking modifications that are best done at a lower level in the search engine.

How to Implement Business Rules in LucidWorks Search

There are two main areas to cover for implementing business rules with LucidWorks Search:

First, determine how the rules will be implemented. There are a variety of methods, each described in the section on [Configuring Business Rules in LucidWorks Search](#).

Second, define the rules themselves. LucidWorks Search has integrated Drools, and you'll want to look at the section on [Writing Rules](#) for information on how to construct a rules file.

There are [Example Rules and Recipes](#). If you're not using rules at all, you can [disable business rules](#).

Integrating with your Rules Engine

If you already have a rules engine (such as ILOG's JRules or Fair Isaac's Blaze Advisor) you can hook them into LucidWorks by implementing a RulesEngine class that talks to your rules engine. Naturally, you can also implement your own SearchComponent, DocTransformer, UpdateRequestProcessor, etc., if the ones shipped with LucidWorks do not meet your needs.

Configuring Business Rules in LucidWorks Search

While business rules in LucidWorks Search is based on the add-on Solr module of the same name, LucidWorks Search is configured out of the box to use rules files. There are several points of configuration that can be modified or re-used as needed, and includes:

- a requestHandler named `"/rulesMgr"`
- a searchComponent named `"landingPage"`
- a searchComponent named `"firstRulesComp"`
- a searchComponent named `"lastRulesComp"`
- addition of rules to the `updateRequestProcessorChain` named `"lucid-update-chain"`
- a document transformer named `"rules"`

There are also a few [optional requestHandlers](#) that could be configured if desired.

The rest of this section will describe each one, and discuss how to integrate it with an existing Solr system. If you are not yet familiar with requestHandlers, searchComponents and similar configurations in a `solrconfig.xml` file, you may want to review the Solr Reference Guide section [RequestHandlers and SearchComponents in SolrConfig](#).

RequestHandlers

The `RulesEngineManagerHandler` is the Solr `requestHandler` that holds on to references to the various rules engine instances specified in the Solr configuration. The manager maintains a map of engines to their names. Most components are set up to take in the name of this `RequestHandler` and then go ask it for the engine by name.

`/rulesMgr`

The `rulesMgr` handles references to rules engine instances. Each of the engines are defined and used by the searchComponents.

Topics discussed in this section:

- [RequestHandlers](#)
 - [/rulesMgr](#)
 - [Optional RequestHandlers](#)
- [SearchComponents](#)
 - [firstRulesComp](#)
 - [lastRulesComp](#)
 - [Rules Component Parameters](#)
 - [landingPage](#)
- [UpdateRequestProcessorChain](#)
- [Document Transformer](#)
- [Rules with Index Replication](#)


```
<requestHandler class="com.lucid.rules.RulesEngineManagerHandler" name="/rulesMgr">
  <!-- Engines can be shared, but they don't have to be.  A SearchComponent or other
consumer can
    specify the engine they want by name.
  -->
  <lst name="engines">
    <lst name="engine">
      <str name="name">first</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultFirst.drl</str>
      </lst>
    </lst>
    <lst name="engine">
      <str name="name">landing</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultLanding.drl</str>
      </lst>
    </lst>
    <!-- Engine is using rules that are designed to be called after all other
components -->
    <lst name="engine">
      <str name="name">last</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultLast.drl</str>
      </lst>
    </lst>
    <lst name="engine">
      <str name="name">docs</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultDocs.drl</str>
      </lst>
    </lst>
  </lst>
</requestHandler>
```

Optional RequestHandlers

The following requestHandlers are not included with LucidWorks Search by default, but could be added to the `solrconfig.xml` for a collection. Much of the same functionality exists with the default `/lucid` requestHandler, but these might be useful if you would like to have specific handlers for specific purposes. Some of the example rules files reference these handlers.

/update-with-rules

This is an `updateRequestHandler` for indexing documents. Note that it calls the `updateRequestProcessorChain`, defined later. This allows using rules to alter documents while they are being indexed, using Solr's standard `updateRequestHandler` class.

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

The `/update-with-rules` requestHandler works in a similar way to the default `/update` requestHandler and takes the same parameters when used. As with the default `/update` requestHandler, in Solr 4.x versions, you can use this one handler to send documents to Solr as CSV, JSON, and XML files.

/update-extract-with-rules

This is another `updateRequestHandler` for indexing documents with rules, and it also calls the `updateRequestProcessorChain`. However, this requestHandler is based on Solr's `ExtractingRequestHandler`, which allows you to use Tika to extract content from complex files such as Word documents, PDF files, and binary files.

```
<requestHandler name="/update-extract-with-rules"
  startup="lazy"
  class="solr.extraction.ExtractingRequestHandler" >
  <lst name="defaults">
    <str name="update.chain">update-with-rules-chain</str>
    <str name="lowernames">true</str>
    <str name="uprefix">ignored_</str>
    <!-- capture link hrefs but ignore div attributes -->
    <str name="captureAttr">true</str>
    <str name="fmap.a">links</str>
    <str name="fmap.div">ignored_</str>
  </lst>
</requestHandler>
```

Because this requestHandler is based on the `ExtractingRequestHandler`, it allows the same parameters.

/search-with-rules

This is a requestHandler which provides an example rules-based search. Note in the configuration below that we have defined two arrays, `"first-components"` and `"last-components"` and named specific searchComponents.

```
<requestHandler name="/search-with-rules" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="df">text</str>
  </lst>
  <arr name="first-components">
    <str>landingPage</str>
    <str>firstRulesComp</str>
  </arr>
  <arr name="last-components">
    <str>lastRulesComp</str>
  </arr>
</requestHandler>
```

If you want to integrate rules with an existing requestHandler, you can add the named searchComponents to the handler, in the same way shown in this example.

SearchComponents

The primary mechanism for applying rules at query time (i.e., not a document indexing request) is via a Solr `searchComponent` called `RulesComponent`. The `RulesComponent` can be configured to occur anywhere in the searchComponent, but it is typically best to configure it to be the first item in the chain after the filter by role component, since it is often the case that you want rules to make decisions based on the application's input parameters (such as the query, sort, etc.) and you want the rules to make changes before they get processed by the other components. For instance, you may have a rule that fires when the user query is equal to "title:dogs" and you want the rule to change the query to be "title:dogs AND category:pets". By configuring the component first in the chain, you will be able to change the query before it is parsed, thus saving extra rule writing involving re-arranging complex Query objects.

firstRulesComp

The `firstRulesComp` is a searchComponent which is meant to be placed within the "first-components" capability of Solr. This allows applying a rule before other searchComponents have been applied. An example of this might be to limit search results with parameters not entered by the user (which may be conditional depending on the user, or other factors). Then other searchComponents, such as faceting or highlighting, can be applied to the reduced result set.

```
<searchComponent class="com.lucid.rules.RulesComponent" name="firstRulesComp">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">first</str>
  <!-- The handle can be used to turn on or off explicit rules components in the
       case when you have multiple rules at different stages of the component
  ordering-->
  <str name="handle">first</str>
</searchComponent>
```

lastRulesComp

The lastRulesComp is a searchComponent which is meant to be placed within the "last-components" capability of Solr. This allows applying a rule after other searchComponents have been applied.

```
<searchComponent class="com.lucid.rules.RulesComponent" name="lastRulesComp">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">last</str>
  <str name="handle">last</str>
</searchComponent>
```

Rules Component Parameters

Input Parameters

There is a fair amount of control around exactly when rules will be fired.

Parameter	Type	Description	Default	Example
rules	boolean	Turn on or off the RulesComponent	false	&rules=false
rules.<handle name>	boolean	Turn on or off a specific RulesComponent instance using the handle name	true	&rules.first=false
rules.prepare	boolean	Turn off rule processing as part of the prepare phase	true	&rules.prepare=false
rules.process	boolean	Turn off rule processing as part of the process phase	true	&rules.process=false
rules.finishStage	boolean	Turn off rule processing as part of the finishStage phase	true	&rules.finishStage=false

The system does not currently allow you to turn off individual phases of an instance (unless it is the only instance that is configured). In other words, if two RulesComponent-s are configured, it is not possible to turn off the process phase of only one.

Facts Collected for the RulesComponent

The facts collected for the RulesComponent are:

- The ResponseBuilder object
- The SolrQueryRequest object
- The schema for the index
- The context information of the request (including the phase of processing, like "process" or "prepare")
- The SolrQueryResponse object
- The query response NamedList
- The request parameters map as a ModifiableSolrParams instance (can be edited by rules)

- The generated query object, which is the same as the parsed query. In some cases, clauses of the query will be added to the knowledge session to allow the rules engine to evaluate any part of the query.
- The filter queries
- Response results (the `DocListAndSet` instance)
- The sort spec
- The grouping spec
- Facet counts

Some of the items on this list will only be available to the rules engine if the `RulesComponent` is placed after the associated `searchComponent` for the fact. For example, in order to have facet information available to the rules engine, the `RulesComponent` has to be placed after that component in the `searchComponents` chain for the `requestHandler`.

[Back to Top](#)

landingPage

The `landingPage` `searchComponent` is generally used to define a specific result for conditions that match the rule. For example, you could redirect users to a specific page of the website in response to a query, or you could highlight specific documents for a query in combination with other factors such as time of day, or user attributes.

The `LandingPageComponent` does not turn off other components in the chain, but it is generally possible for the rules engine to do so. For example, if you wanted to disable faceting, you would add a rule such as `facet=false`. For the query, you could add `query=false`. The exact methods you need are dependent on the search components you have enabled. See also the section `Search Components API` for one approach to finding enabled search components for the `requestHandler` in use.

Placing the landing page in the output is also the responsibility of the rule writer. In essence, all the `LandingPageComponent` does is guarantee that it is called as part of rules and fact preparation and that the rules used can be configured separately from other rules.

```
<searchComponent class="com.lucid.rules.LandingPageComponent" name="landingPage">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">landing</str>
  <!-- The handle can be used to turn on or off explicit rules components in the
        case when you have multiple rules at different stages of the component
ordering
  -->
  <str name="handle">landing</str>
</searchComponent>
```

Input Parameters

Like the `RulesComponent`, the `LandingPageComponent` has several parameters. One thing to note is that the `LandingPageComponent` is only executed in the prepare phase of rules execution, so other available parameters will likely not be required for your implementation.

Parameter	Type	Description	Default	Example
landing	boolean	Turn on or off the <code>LandingPageComponent</code>	false	<code>&landing=false</code>
landing.<handle name>	boolean	Turn on or off a specific <code>LandingPageComponent</code> instance using the handle name	true	<code>&landing.first=false</code>
landing.prepare	boolean	Turn off rule processing as part of the prepare phase	true	<code>&landing.prepare=false</code>
landing.process	boolean	Turn off rule processing as part of the process phase	true	<code>&landing.process=false</code>
landing.finishStage	boolean	Turn off rule processing as part of the finishStage phase	true	<code>&landing.finishStage=false</code>

Facts Collected for the `LandingPageComponent`

The facts collected for the `LandingPageComponent` are:

- The `ResponseBuilder` object
- The `SolrQueryRequest` object
- The schema for the index
- The context information of the request (including the phase of processing, like "process" or "prepare")
- The `SolrQueryResponse` object
- The query response `NamedList`
- The request parameters map as a `ModifiableSolrParams` instance (can be edited by rules)
- The generated query object, which is the same as the parsed query. In some cases, clauses of the query will be added to the knowledge session to allow the rules engine to evaluate any part of the query.
- The filter queries
- Response results (the `DocListAndSet` instance)
- The sort spec
- The grouping spec
- Facet counts

Some of the items on this list will only be available to the rules engine if the `{ LandingPageComponent }` is placed after the associated `searchComponent` for the fact. For example, in order to have facet information, the `LandingPageComponent` has to be placed after that component in the `searchComponents` chain for the `requestHandler`.

[Back to Top](#)

UpdateRequestProcessorChain

LucidWorks supplies a custom updateRequestProcessorChain called "lucid-update-chain". We have added the RulesUpdateProcessor to the default chain. This allows you to make transformations to documents while they are being indexed. Note that the example "/update-with-rules" and "/update-extract-with-rules" requestHandlers both call this chain definition.

By default, the RulesUpdateProcessor is configured in the lucid-update-chain and can be enabled or disabled by passing in the name of the handle, prefixed by rules.. For instance, if the Processor has a handle of docProc, then &rules.docProc=false would disable the processor and processing would continue down the chain. Rule processing is on by default.

Like the query-related rules processing, altering documents relies on facts during the knowledge session.

Here is the default configuration for the lucid-update-chain in the solrconfig.xml file for each collection:

```
<updateRequestProcessorChain name="lucid-update-chain">
  <processor class="com.lucid.update.CommitWithinUpdateProcessorFactory" />
  <processor class="com.lucid.update.FieldMappingUpdateProcessorFactory" />
    <processor class="com.lucid.rules.RulesUpdateProcessorFactory">
      <str name="requestHandler">/rulesMgr</str>
      <!-- we re-use the engine, but we could have an independent one-->
      <str name="engine">docs</str>
      <!-- Each one should have it's own handle, as you can have multiple
in the chain -->
      <str name="handle">docProc</str>
    </processor>
  <processor class="com.lucid.update.DistributedUpdateProcessorFactory">
    <!-- example configuration... "shards should be in the *same* order for
every server in a cluster. Only "self" should change to represent what
server
    *this* is. <str name="self">localhost:8983/solr</str> <arr name="shards">
      <str>localhost:8983/solr</str> <str>localhost:7574/solr</str> </arr> -->
    </processor>
  <processor class="solr.LogUpdateProcessorFactory">
    <int name="maxNumToLog">10</int>
  </processor>
  <processor class="solr.DistributedUpdateProcessorFactory" />
  <processor class="solr.RunUpdateProcessorFactory" />
</updateRequestProcessorChain>
```

To disable rules processing, you can either remove or comment out the section that defines the com.lucid.rules.RulesUpdateProcessorFactory parameters.

Facts Collected for the RulesUpdateProcessor

The facts collected for the `RulesUpdateProcessor` are:

- The `AddUpdateCommand` as received in the `UpdateRequestProcessor.processAdd(AddUpdateCommand)` method
- The `SolrInputDocument` being added
- The schema for the index

[Back to Top](#)

Document Transformer

The document transformer allows applying rules that alter documents during query time. It is invoked as part of Solr's response and can inject or modify fields before they are returned.

```
<transformer name="rules" class="com.lucid.rules.RulesDocTransformerFactory">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">docs</str>
</transformer>
```

Note that alterations to documents made with this transformer are not saved to the documents themselves. If you want to make changes that are saved with documents, use the `UpdateRequestProcessorChain` instead.



Altering a field will not cause an item to be resorted

If, for example, you are sorting by price and you change one of the document's prices, this will not cause a re-sort. If you want to do that, we suggest you use Solr's Sort by Function capability.

Facts Collected for the `RulesDocTransformer`

The facts collected for the `RulesDocTransformer` are:

- The `SolrInputDocument` being transformed
- The `docId` of the document being transformed (the Lucene internal `docId`, not Solr's `uniqueKey`)
- The schema for the index

[Back to Top](#)

Rules with Index Replication

If you are using what is now considered "old-style" replication (i.e., you are not using SolrCloud), you should add the rules files to the `confFiles` list of configuration files that are copied to the slave servers with each update.


```
<!-- Optional -->
<!-- If using older v3 style master/slave replication, instead of 4x SolrCloud,
      add these files to your master confFiles list
      <str
name="confFiles">...,rules/defaultFirst.drl,rules/defaultLast.drl,rules/defaultLanding.drl
<requestHandler name="/replication" class="solr.ReplicationHandler" >
    <lst name="master">
        <str name="replicateAfter">commit</str>
        <str name="replicateAfter">startup</str>
    </lst>
name="confFiles">schema.xml,stopwords.txt,rules/defaultFirst.drl,rules/defaultLast.drl,rul
</lst>
    <lst name="slave">
        <str name="masterUrl">http://your-master-hostname:8983/solr</str>
        <str name="pollInterval">00:00:60</str>
    </lst>

</requestHandler>
```

[Back to Top](#)

Writing Rules

The Business Rules module integrates [Drools 5.5](#) with LucidWorks Search. The Drools [Rule Language Reference](#) provides a much more thorough overview, but the below can serve as a brief introduction.

In Drools, rules are defined with Java-like declarations. While the software is meant to be easier for non-programmers to write rules, it is still a heavily technical syntax and assumes some technical proficiency.

To help you with writing rules, we have provided a `DroolsHelper.java` class which consists of helper functions to make the task easier. You can find this class in the `solr-business-rules.jar` file (the full name may include version numbers, but you should only have one `.jar` starting with `solr-business-rules`) found in `$LWS_HOME/app/webapps/lwe-core/lwe-core/WEB-INF/lib`. It is also included below.

In this section:

- [Rules Files](#)
- [Rule Declarations](#)
 - [rule and Attributes](#)
 - [when Conditions](#)
 - [then Actions](#)
- [DroolsHelper Class](#)
 - [Limitations](#)
- [Related Topics](#)

Rules Files

A rules file has a file extension of `.drl`. For the Business Rules module, we have placed the rules in the `conf` directory of each Solr collection, in a sub-directory called `rules`. The example configurations assume this path; if they are located in another area of the filesystem, the examples will need to be updated.

Before starting the rule declarations, the package is defined, as are any imports and globals. The import statements are similar to import statements in Java, where you specify the fully qualified paths and type names for objects that will be used with the rules. The global statements allow you to make application objects available to the rules, such as if there is data or services the rules use.

Rule Declarations

At it's simplest, a rule declaration looks like this:

```
rule "name"
<a set of attributes>
when
<a set of qualifying conditions, in Drools called "Left Hand Side">
then
<a set of actions to perform, in Drools called "Right Hand Side">
end
```

rule and Attributes

The first step is to state you are going to define a rule, simply with `rule` and a name of the rule.

Next, you can define attributes for the rule, which influence the behavior of the rule. One of the most important of these is `no-loop`, which prevents an infinite loop if a rule modifies a fact that causes the rule to activate again. There are several other attributes, however, which may be important to your rule. See the Drools documentation on [Rule Attributes](#) for more information.

when Conditions

In Drools language, the conditions that must be met for a rule to fire are also called "Left Hand Side".

Conditions work on one or more patterns, which include the object and constraints. For example, a condition like `$rb: ResponseBuilder($qStr : req.params.get("q") matches "(?i).*ipod.*"))` will match queries sent to Solr containing the term "ipod". What's going on in this example?

First, we've declared that the variable `$rb` will match the object `ResponseBuilder`. The `ResponseBuilder` is a Solr class that builds the query responses. The rest of the condition states we want to look at what the value was for Solr's `q` parameter, and match queries that contain the term "ipod".

There are multiple variations on how to declare the conditions. You can use Java expressions, booleans, binding variables, maps, and many more. Refer to the Drools documentation on [Left Hand Side \(when\) syntax](#) for all of the options and details on how to use them.

then Actions

In Drools language, the actions of a rule are also called "Right Hand Side". These are the changes that should be made to the "facts" known to the rules engine. In search, this would be changes to documents, the order of results, or other impacts on the results of the user's query. Keep in mind that these actions should not be conditional (as in, "when this, maybe this"), but atomic, meaning all of the stated actions should be performed (as in, "when this, then this"). If you find you need further conditions, you may want to consider breaking your rule into smaller pieces to achieve this goal.

As with `when` conditions, there are multiple variations on how to use `then` actions. Of particular assistance here is the `DroolsHelper.class`, found in the `solr-business-rules-0.1-solr-4.4.0.jar`, where several methods have been pre-defined such as `addToResponse`, which allows adding a key-value pair to the response, and `modRequest`, which modifies the request to Solr.

Refer to the Drools documentation on [Right Hand Side \(then\)](#) for more details.

[Back to Top](#)

DroolsHelper Class

The `DroolsHelper` class contains a number of methods that can be invoked by rules writers to help with common tasks and simplify the "then" part of the rule. For instance, there is a method that can take in a query and a boost and set the boost value. There are also methods for helping merge separate facet requests together (such as a field facet with a facet query). For instance, it has methods that evaluate what phase the engine is in and returns true or false if it matches an expected value. This can be useful if you want rules to fire only during certain phases of the `SearchComponent` process (i.e. prepare, process, etc.). To see this in action, notice the use of the `hasPhaseMatch()` method in the [example rules section](#).

The `DroolsHelper.class` file may not be in your distribution of LucidWorks. For that reason, we've provided the text of the code below.

```
package com.lucid.rules.drools;

public class DroolsHelper extends java.lang.Object
{
    /* Fields */
    private static transient org.slf4j.Logger log;
    public final static java.lang.String RULES_PHASE = "rulesPhase";
    public final static java.lang.String RULES_HANDLE = "rulesHandle";

    /* Constructors */
    public DroolsHelper() {
    }

    /* Methods */
    public static boolean
    hasPhaseMatch(org.apache.solr.handler.component.ResponseBuilder, java.lang.String) {
    }

    public static boolean
    hasPhaseMatch(org.apache.solr.handler.component.ResponseBuilder, java.lang.String,
    java.lang.String) {
    }

    public static boolean
    hasHandlerNameMatch(org.apache.solr.handler.component.ResponseBuilder,
    java.lang.String) {
    }

    public static void boostQuery(org.apache.lucene.search.Query, float) {
    }

    public static void addToResponse(org.apache.solr.handler.component.ResponseBuilder,
    java.lang.String, java.lang.Object) {
    }

    public static void addToResponse(org.apache.solr.common.util.NamedList,
```

```
java.lang.String, java.lang.Object) {  
    }  
  
    public static void mergeFacets(org.apache.solr.handler.component.ResponseBuilder,  
java.lang.String, int, java.lang.String[]) {  
    }  
  
    public static void addFacet(org.apache.solr.handler.component.ResponseBuilder,  
java.lang.String, java.lang.String, int, int) {  
    }  
  
    public static void modRequest(org.apache.solr.handler.component.ResponseBuilder,  
java.lang.String, java.lang.String[]) {  
    }  
  
    public static void modRequest(org.apache.solr.handler.component.ResponseBuilder,  
java.lang.String, int) {  
    }  
  
    public static void modRequest(org.apache.solr.handler.component.ResponseBuilder,  
java.lang.String, boolean) {  
    }  
  
    public static boolean contains(java.lang.String, java.lang.String) {  
    }  
  
    public static java.util.Collection analyze(org.apache.solr.schema.IndexSchema,  
java.lang.String, java.lang.String) throws java.io.IOException {  
    }  
  
}
```

Limitations

Since the implementation is stateless, there is obviously no way to write rules that go across requests without implementing your own RulesEngine.

[Back to Top](#)

Related Topics

There are several rules provided as examples, which may help you get started with the rules language. See Example Rules for a walk-through of two examples, plus an overview of other included examples.

Example Rules and Recipes

Several example rules are provided in the `$app/examples/business_rules` directory of your LucidWorks Search installation.

In this section we'll pick a couple of the rules and walk through them.

Sample Rule Files

The example rules are designed to be used with the example documents provided by Solr. Each file includes extensive comments that explain what they are doing and how to use them with the sample documents that are included with Solr. Note, however, that LucidWorks Search does not include the same directory of sample documents, and the default LucidWorks Search `schema.xml` is also different. These rules may need a bit of tweaking to work correctly with your own content and customized schema.

In most cases, the recommendation is to add new rules to the files in the `rules` directory found in the `$LWS_HOME/conf/solr/cores/collection/conf` directory, where `collection` is the name of the collection where rules will be used.

While it's possible to define multiple rules files in `solrconfig.xml` (in the `/rulesMgr requestHandler` section, it is simpler to use a single rules file (when possible) for each rules engine. This keeps all your rules in one place, making them easier to manage. You can modify the name of the single file if you'd like, just be sure to update the `/rulesMgr requestHandler` appropriately.

The following rules are included as examples:

Filename	Rule Type	What It Does
<code>defaultDocs-create-title.drl</code>	Indexing rule	Adds title fields to incoming documents.
<code>defaultDocs-manufacturer-check.drl</code>	Indexing rule	Copies the document ID field to the <code>manu</code> field on documents where <code>manu</code> is blank.
<code>defaultDocs-price-check.drl</code>	Indexing rule	Checks the price of an incoming document and adds a label when it matches a specific criteria. This approach is designed for times when using text (i.e., JSON, XML) codecs for indexing.
<code>defaultDocs-price-check-long-form.drl</code>	Indexing rule	An alternate approach to price checks. This approach is designed for times when using binary (i.e., Javabin) codecs for indexing.

defaultFirst-apple.drl	Query rule	Adds a defined manu field to all searches for a specific term.
defaultFirst-facets-part1of2.drl	Query rule	First of two steps to modify a facet; injects a facet query and alters the facet limit.
defaultFirst-from-readme-file.drl	Query rule	Adds a term to the query.
defaultFirst-model-number.drl	Query rule	Defines a method to find model numbers in a query, and if found looks in the ID field for a match.
defaultFirst-show-phases.drl		Demonstrates the phases of filtering.
defaultLanding-belkin.drl	Landing rule	Returns a specific URL in response to a query, which can be used by the front-end to either redirect the user or display it a specific way.
defaultLast-facets-part2of2.drl	Query rule	Part two of the earlier rule to modify a facet; injects the facet to the response.

Detailed Examples

README Example

This example is included in the file `defaultFirst-from-readme-file.drl`. The goal of this rule is to add query terms to a search when the user enters a specific string.

First, here is the text of the rule (note, this isn't the whole file, just the part that defines a rule; be sure to look at the whole rule for important comments on how to run it).

```
rule "electronics"
no-loop
when
  $rb: ResponseBuilder($qStr : req.params.get("q") == "text:electronics");
then
  addToResponse($rb, "origQuery", $qStr);
  addToResponse($rb, "modQuery", "text:electronics text:apache");
  modRequest($rb, "q", "text:electronics text:apache");
end
```

Let's step through this example in detail.

Line 1 states we are declaring a rule and gives it the name "electronics".

Line 2 says to only run the rule once to prevent an infinite loop. In this case, our `when` statement looks for the query term "electronics" on the field "text"; after the modifications from the rule, the query will still match the rule, which could make it fire again. Using `no-loop` prevents the rule firing over and over.

Line 3 starts the `when` conditions.

Line 4 uses Solr's `ResponseBuilder` to analyze the query, and match when the query (in the `q` parameters of the request sent to Solr) matches "text:electronics". Note this line is also setting a variable `$qStr`, and assigning it the query and parameters. This variable will be used again later.

Line 5 starts the `then` actions.

Line 6 defines a key/value pair for the `ResponseBuilder` of "origQuery" and the query string variable defined in line 4 (`$qStr`).

Line 7 defines another key/value pair for the `ResponseBuilder` of "modQuery", and the modified query string.

Line 8 modifies the request to the `ResponseBuilder` with a key/value pair, modifying the user's entry to include "text:apache" as well as what was initially entered.

Line 9 ends the rule.

To run this rule, once the rule has been added to `rules/defaultFirst.drl`, you can send a request to Solr that looks something like this:

```
http://localhost:8888/solr/collection1/lucid?q=text:electronics&rules=true&rules.first=tr
```

The request should be customized for your hostname and port, and this example also assumes you have indexed Solr's sample documents in the `example/exampledocs` directory.

Landing example

This example is included in the file `defaultLanding-belkin.drl`. The goal of this rule is to force Solr to return a document first in the list when a specific manufacturer ("Belkin") is entered by the user.

First, here is the text of the rule (note, this isn't the whole file, just the part that defines a rule; be sure to look at the whole rule for important comments on how to run it).


```
rule "Landing Page"
no-loop
when
    $rb: ResponseBuilder($qStr : req.params.get("q") == "manu:Belkin");
then
    addToResponse((NamedList)$rb.rsp.getValues().get("responseHeader"), "landingPage",
"http://www.Belkin.com");
end
```

This rule is quite simple, actually, but let's step through it line-by-line.

Line 1 states we are declaring a rule and gives it the name "Landing Page".

Line 2 says to only run the rule once to prevent an infinite loop. In this case, our `when` statement looks for the query term "Belkin" on the field "manu"; after the modifications from the rule, the query will still match the rule, which could make it fire again. Using `no-loop` prevents the rule firing over and over.

Line 3 starts the `when` conditions.

Line 4 uses Solr's `ResponseBuilder` to analyze the query, and match when the query (in the `q` parameters of the request sent to Solr) matches "manu:Belkin". Note this line is also setting a variable `$qStr`, and assigning it the query and parameters. This variable will be used again later.

Line 5 starts the `then` actions.

Line 6 defines a key/value pair to the `NamedList`. In this case, inserting "landing page" and the URL into the `responseHeader`.

Line 7 ends the rule.

Note that this rule by itself does not magically redirect the user to the Belkin website - it includes the information to the client, which then must decide what to do: redirect the user, make it the first result in the list, or some other transformation as needed.

To run this rule, once the rule has been added to `rules/defaultLanding.drl`, you can send a request to Solr that looks something like this:

```
http://localhost:8888/solr/collection1/lucid?q=manu:Belkin&rules=true&landing=true
```

The request should be customized for your hostname and port, and this example also assumes you have indexed Solr's sample documents in the `example/exampledocs` directory.

Disabling Business Rules

Business rules are enabled by default. Even if you are not using rules, there should be no impact on performance, but if you want to simplify your configuration, you can remove or comment out references to rules in the `solrconfig.xml` file for each collection.

It would be possible to remove these rules parameters from the default `solrconfig.xml` file and create a template for future collection creation. To learn more about this, see the section on Collection Templates.



When removing business rules from the `solrconfig.xml` file, LucidWorks will need to be either stopped while making the changes, or restarted once the changes are made.

These are the steps to disabling business rules:

- [Remove Rules from Update Chain](#)
- [Remove Rules from the /lucid Request Handler](#)
- [Remove the Rules Request Handler](#)
- [Remove Rules Search Components](#)
- [Remove the RulesDocTransformer](#)
- [Remove Rules From the Replication Handler](#)

Remove Rules from Update Chain

Comment out the section that defines the Rules Update Processor (`<processor class="com.lucid.rules.RulesUpdateProcessorFactory">` until the closing `</processor>` tag).

In most cases, this is sufficient to disable business rules. However, the next sections will assist you in fully removing business rules from your implementation.

```
<updateRequestProcessorChain name="lucid-update-chain">
  <processor class="com.lucid.update.CommitWithinUpdateProcessorFactory"/>
  <processor class="com.lucid.rules.RulesUpdateProcessorFactory">
    <str name="requestHandler">/rulesMgr</str>
    <!-- we re-use the engine, but we could have an independent one-->
    <str name="engine">docs</str>
    <!-- Each one should have it's own handle, as you can have multiple in
the chain -->
    <str name="handle">docProc</str>
  </processor>
  <processor class="com.lucid.update.DistributedUpdateProcessorFactory">
    <!-- example configuration... "shards should be in the *same* order for
every server in a cluster. Only "self" should change to represent what server
*this* is. This is only used for Index Replication.
    <str name="self">localhost:8983/solr</str> <arr name="shards">
    <str>localhost:8983/solr</str> <str>localhost:7574/solr</str> </arr> -->
  </processor>
  <processor class="solr.LogUpdateProcessorFactory">
    <int name="maxNumToLog">10</int>
  </processor>
  <processor class="solr.DistributedUpdateProcessorFactory"/>
  <processor class="com.lucid.update.FieldMappingUpdateProcessorFactory"/>
  <processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

The specific section to remove is:

```
<processor class="com.lucid.rules.RulesUpdateProcessorFactory">
  <str name="requestHandler">/rulesMgr</str>
  <!-- we re-use the engine, but we could have an independent one-->
  <str name="engine">docs</str>
  <!-- Each one should have it's own handle, as you can have multiple in the
chain -->
  <str name="handle">docProc</str>
</processor>
```

[Back to Top](#)

Remove Rules from the /lucid Request Handler

Find the section as below that defines the /lucid request handler, and remove the lines for landingPage and firstRulesComp and lastRulesComp.

```
<requestHandler class="solr.StandardRequestHandler" name="/lucid">
  <arr name="components">
    <str>filterbyrole</str>
    <str>landingPage</str>
    <str>firstRulesComp</str>
    <str>query</str>
    <str>mlt</str>
    <str>stats</str>
    <str>feedback</str>
    <!-- Note: highlight needs to be after feedback -->
    <str>highlight</str>
    <!-- Note: facet also needs to be after feedback -->
    <str>facet</str>
    <str>spellcheck</str>
    <str>lastRulesComp</str>
    <str>debug</str>
  </arr>
  ...
</requestHandler>
```

[Back to Top](#)

Remove the Rules Request Handler

The rules request handler defines the `RuleEngine` instances and the rules files. The entire section copied below can be removed or commented out.

```
<requestHandler class="com.lucid.rules.RulesEngineManagerHandler" name="/rulesMgr">
  <!--
    Engines can be shared, but they don't have to be. A SearchComponent or other
    consumer can
    specify the engine they want by name.
  -->
  <lst name="engines">
    <lst name="engine">
      <str name="name">first</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultFirst.drl</str>
      </lst>
      <!-- The fact collector defines what facts get injected into the rules engines
working memory -->
      <!--<lst name="factCollector">
        <str name="class">com.lucid.rules.drools.FactCollector</str>
      </lst>-->
    </lst>

    <lst name="engine">
      <str name="name">landing</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultLanding.drl</str>
      </lst>
    </lst>

    <!-- Engine is using rules that are designed to be called after all other
components -->
    <lst name="engine">
      <str name="name">last</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultLast.drl</str>
      </lst>
    </lst>
    <lst name="engine">
      <str name="name">docs</str>
      <str
name="class">com.lucid.rules.drools.stateless.StatelessDroolsRulesEngine</str>
      <lst name="rules">
        <str name="file">rules/defaultDocs.drl</str>
      </lst>
    </lst>
  </lst>
</requestHandler>
```

[Back to Top](#)

Remove Rules Search Components

The search components allow the rules to make changes to queries, based on the rules defined. The entire sections shown below can be removed or commented out.

```
<searchComponent class="com.lucid.rules.LandingPageComponent" name="landingPage">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">landing</str>
  <!-- The handle can be used to turn on or off explicit rules components in the
       case when you have multiple rules at different stages of the component ordering
  -->
  <str name="handle">landing</str>
</searchComponent>
<searchComponent class="com.lucid.rules.RulesComponent" name="firstRulesComp">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">first</str>
  <!-- The handle can be used to turn on or off explicit rules components in the
       case when you have multiple rules at different stages of the component ordering-->
  <str name="handle">first</str>
</searchComponent>
<searchComponent class="com.lucid.rules.RulesComponent" name="lastRulesComp">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">last</str>
  <str name="handle">last</str>
</searchComponent>
```

[Back to Top](#)

Remove the RulesDocTransformer

The RulesDocTransformer allows business rules to inject or modify fields in a document before returning them to a client.

```
<transformer class="com.lucid.rules.RulesDocTransformerFactory" name="rules">
  <str name="requestHandler">/rulesMgr</str>
  <str name="engine">docs</str>
</transformer>
```

[Back to Top](#)

Remove Rules From the Replication Handler

If using Index Replication, remove the rules-related files from the list of `conf` files to replicate between servers. In this section:

```
<requestHandler class="solr.ReplicationHandler" name="/replication">
  <lst name="master">
    <str name="replicateAfter">commit</str>
    <str
name="confFiles">admin-extra.html,admin-extra.menu-bottom.html,admin-extra.menu-top.html,e
  </lst>
</requestHandler>
```

Specifically, remove `rules/defaultFirst.drl`, `rules/defaultLast.drl`,
`rules/defaultLanding.drl`, and `rules/defaultDocs.drl`.

[Back to Top](#)

Security and User Management

Generally, enterprise-level application designers must take into account four main security considerations for any search application:

- Network access to the various components of the service
- Authentication of users
- Authorization to use various parts of the user interface
- Authorization to view certain documents

LucidWorks Search implements security for each of these as follows:

Network access: Because the [components of LucidWorks](#) (LWE-Core, LWE-UI and LWE-Connectors) run on different ports, an administrator can easily secure individual components at the network level by restricting access to the port in question. For example, if only the Admin and Search UI services need to be accessible outside the production network, an administrator can leave those ports open while blocking LWE-Core. The chapter [Securing LucidWorks](#) describes this process in more detail. Note that if you are using the LucidWorks Search document authorization features this step is particularly important, as direct access to the underlying Solr application can circumvent these measures.

In addition, you may want to ensure that the components use SSL for communication or that users access the Admin UI via HTTPS. The chapter [Enabling SSL](#) describes how to do that in more detail.

User authentication: LucidWorks supports LDAP binding for user authentication, so an administrator can create roles or groups on an external LDAP server, then use them to control access to UI functionality or sets of documents. The chapter [LDAP Integration](#) describes how to configure LDAP for LucidWorks.

UI authorization: LucidWorks controls access to the Admin UI and the Search UI. The chapter [LDAP Integration](#) discusses how to configure these access levels in order to give different LDAP users and groups authorization to use these different functions.

Document authorization: LucidWorks allows the administrator to configure document filters for different roles. These document filters then limit what documents appear in search results for users in those roles. For example, the administrator can create a filter that enables users in the *finance* role to see only documents that satisfy a query of *department:finance*. You can create these filters with the Search Filters screen of the Admin UI. LucidWorks also enables the creation of document-based filtering, in which only the owner (or owners) of a document are able to see it. The section [Restricting Access to Content](#) describes how to set up your documents to support this functionality.

Securing LucidWorks

There are several approaches to securing access to LucidWorks Search: by requiring authentication to access the UIs and APIs, by restricting users to specific roles within the system, and finally by restricting access to certain documents in results lists.

Restricting Access

LucidWorks Search consists of [three components](#): LWE-Core, LWE-UI, and LWE-Connectors.

Because it provides access to the REST API, direct access to the LWE-Core component provides access to all of Solr's capabilities, including adding and removing documents, retrieving stored field values for all documents, and additional LucidWorks Search-enhanced capabilities such as job scheduling and system status. The LWE-Core component should only be directly HTTP accessible to other components that need access to Solr or REST API interfaces. If you are using a single server installation and don't want to expose Solr or REST API interfaces via the network then you might want to restrict access to LWE-Core to localhost only. You can do that by adding the [socket connector's host attribute for the Jetty container](#).

Topics covered in this section:

- [Restricting Access](#)
 - [Enabling Basic Auth for UIs and APIs](#)
- [Restricting Access to LucidWorks Search User Interfaces](#)
- [Hiding Documents by Restricting Access](#)
- [Related Topics](#)

You can also restrict direct access to LucidWorks components by IP address, or by fronting it with an authenticating firewall. For a production implementation, consider restricting access to the component HTTP ports to only those required by the application, just as one would do with a typical relational database. If you are using the built-in search filters or document-level authentication, you **must** prevent access to LucidWorks by any process other than your application in order to prevent circumvention of these features.



Implementing SSL

Each of the components can be implemented with SSL. See the chapter [Enabling SSL](#) for more details.

Enabling Basic Auth for UIs and APIs

It is additionally possible to require basic authentication before accessing the LucidWorks Search UIs (Admin and Search UIs) and REST APIs. This entails creating a `realm.properties` file that contains usernames and passwords, then configuring the `jetty.xml` files to use `realm.properties`, and finally modifying the `web.xml` file for each interface to be restricted. This does not replace the built-in user authentication for LucidWorks Search (i.e., the login to access the UIs), but adds an additional layer of authentication and authorization.

Because LucidWorks Search components run in separate JVMs, they run in separate Jetty containers. However, you should secure both the LWE-UI and LWE-Core components so they can successfully communicate with one another. The LWE-Connectors JVM does not need authentication, since it generally only needs to communicate with the LWE-Core component internally.

Modify `jetty.xml`

The `jetty.xml` file contains a sample configuration that is commented out. This sample can be used by removing the comment markers and changing the `name` parameter as needed. The default uses "Test Auth" for the name, but in the below you'll see we have changed that to "Auth". The name can be whatever you'd like it to be, but it must match the name you use in the `web.xml` file configuration (below).

```
<Call name="addBean">
  <Arg>
    <New class="org.eclipse.jetty.security.HashLoginService">
      <Set name="name">Auth</Set>
      <Set name="config"><SystemProperty name="lucidworksConfHome"
default="."/>/jetty/lwe-core/etc/realm.properties</Set>
      <Set name="refreshInterval">0</Set>
    </New>
  </Arg>
</Call>
```

This configuration also defines the location of the `realm.properties` file, which you will create in the next step. Note that the above example defines a path of `/jetty/lwe-core/etc/etc/realm.properties`. If the user accounts will be the same for both the UIs and the APIs, it is fine to refer to the same file for both components. If, however, the users and/or roles will be different, you need to change the path to the appropriate `realms.properties` file for each component.

These changes need to be done two times: once for the `jetty.xml` file for the LWE-UI component, and again for the `jetty.xml` for the LWE-Core component. These files are found at the following paths:

- `$LWS_HOME/conf/jetty/lwe-core/etc/jetty.xml`
- `$LWS_HOME/conf/jetty/lwe-ui/etc/jetty.xml`

Create a `realm.properties` File

The `realm.properties` file contains usernames, passwords and roles of users who will be allowed to use the UIs and APIs. The passwords can be stored in plain text, encrypted with an MD5 hash, or obfuscated. In this example, we have just used a plain text password:

```
admin: password,user
```

If the password is not defined in plain text, you would preface it with "CRYPT:" if using an MD5 hash or with "OBF:" if obfuscated.

This allows the "admin" user to access the UI and APIs. The role "user" is one that we'll define in the `web.xml` file (described below). If you have multiple roles, they can be listed for each user separated by commas.

Modify web.xml

The `web.xml` file and we'll use it to define the roles, the URLs roles have access to, and the realm name. Below is an example:

```
<!-- Security Constraints -->
<security-role>
  <role-name>user</role-name>
</security-role>
<login-config>
  <realm-name>Auth</realm-name>
</login-config>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>all resources</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
```

In this example, we have defined the `security-role` as "user" and constrained access to all web resources (via the `auth-constraint`) to the role "user". This means users must be defined with the role "user" in `realms.properties`. Additional roles could be defined if needed, but LucidWorks Search already supports "admin" and "search" users (see below), so it may not be necessary to duplicate that functionality.

However, there may be room for roles that restrict access to the APIs. The `url-pattern` could also be modified to support several roles, restricting certain roles to only certain parts of UIs or APIs.

Note that we have defined the `realm-name` as "Auth", which is the same name we used in the `jetty.xml` configuration. Those names must match, or Jetty will not be able to locate the `realms.properties` file.

These changes need to be done two times: once for the `web.xml` file for the LWE-UI component, and again for the `web.xml` for the LWE-Core component. These files are found at the following paths:

- `$LWS_HOME/app/webapps/lwe-core/lwe-core/WEB-INF/web.xml`
- `$LWS_HOME/app/webapps/lwe-ui/WEB-INF/web.xml`

Note that since nearly all of the REST APIs and the Solr Admin UI are powered by the LWE-Core component, specific restrictions for those APIs and Solr Admin UI must be defined in the LWE-Core `web.xml` file. The LWE-UI `web.xml` file can be used to restrict the Admin UI, the Search UI, as well as the Alerts and Users APIs.

Once these changes are completed, LucidWorks Search must be [restarted](#). Additional information about using realms and basic auth with Jetty is available from the [Jetty 8 documentation](#).

Restricting Access to LucidWorks Search User Interfaces

LucidWorks has two built-in authorizations to control user access:

- ADMIN allows users to access any part of the LucidWorks UI.
- SEARCH limits users to only the built-in end user search interface.

You can restrict a user's access to specific parts of the application by mapping manually created or LDAP-supplied usernames and/or LDAP-supplied groups to the appropriate authorization. There are two ways to do this: via the Users API or via the [User screen in the Admin UI](#).

Hiding Documents by Restricting Access

The privileges of the LucidWorks process and the rights that process has to access documents for indexing are crucial to its proper operation. Generally, you want LucidWorks Search to be able to access all documents within a particular folder or from a particular web site. The built-in LucidWorks Search crawlers will index any specified document, as long as the LucidWorks process has permissions to do so. After a document has been indexed, all stored fields are accessible through the Solr interface.

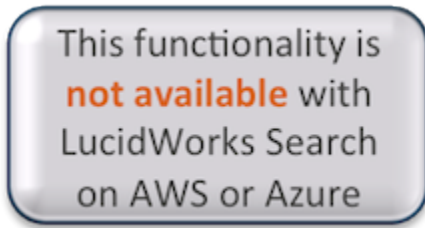
That said, documents can be excluded from indexing by leveraging operating system, file, and web-level security capabilities; if the process doesn't have access, it will not index the content.

Some data sources, such as those configured to crawl content in a [database](#) or in [SMB](#), [SharePoint](#), [S3](#) or [Hadoop over S3](#) servers, credentials need to be supplied for the crawler to access the system. Those credentials determine what documents the crawler has access to. Other data sources may also require credentials to access content.

Related Topics

- [Restricting Access to Content](#)
- [Enabling SSL](#)
- [LDAP Integration](#)

Enabling SSL



Secure Socket Layer (SSL) encryption can be enabled in LucidWorks Search with a few modifications to Jetty configuration files.

- [Steps to Enable SSL](#)
- [Certificate Management](#)
- [Client Certificates for LWE-Core and Connectors](#)
- [Configuring Mutually Authenticated SSL](#)
- [Debugging SSL Configuration](#)
- [Related Topics](#)

Steps to Enable SSL

In the steps below, note that LucidWorks Search components run under Jetty, but have separate configuration files. Each component needs to be enabled separately, although the process for each component is the same. For more information about configuring Jetty to use SSL, see also the Jetty documentation on [Configuring SSL](#).

Step 1: Modify master.conf

If you have already installed LucidWorks Search, you can set these values by modifying the `master.conf` file found in `$LWS_HOME/conf/`. You should change the `address` for each component to include `https`. If you'd like to change the port for each component, that is done in `master.conf` also.

```
# COMPONENT LWE-Core - LWE-Solr + LWE REST API.
# -----
lwecore.enabled=true
lwecore.address=https://127.0.0.1:8888
...
# COMPONENT LWE-Connectors.
# -----
lweconnectors.enabled=true
lweconnectors.address=https://127.0.0.1:8765
...
# COMPONENT LWE-UI - Admin and Search UI as well as Alerts
# -----
lweui.enabled=true
lweui.address=https://127.0.0.1:8989
```

Alternately, each component could be set to `https://` and the desired port during the installation process.

Step 2: Modify jetty.xml for LWE-Core Component

The `jetty.xml` file found in `$LWS_HOME/conf/jetty/lwe-core/etc` needs to be modified to comment out the non-SSL connector. In the file, find the following section and add comment markers at the beginning and at the end (`<!--` and `-->`, respectively):

```
<!--
  <Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.bio.SocketConnector">
        <Set name="port"><SystemProperty name="jetty.port" default="8888"/></Set>
        <Set name="maxIdleTime">50000</Set>
        <Set name="lowResourceMaxIdleTime">1500</Set>
      </New>
    </Arg>
  </Call>
-->
```

Step 3: Modify jetty-ssl.xml for LWE-Core Component

In the directory `$LWS_HOME/conf/jetty/lwe-core/etc` the file `jetty-ssl.xml` should be edited to activate the sample configuration. The configuration is currently commented out, but the comment tags should be removed and the `keyStore`, `keyStorePassword`, `keyManagerPassword`, `trustStore` and `trustStorePassword` properties should be configured.

```

<Configure id="Server" class="org.eclipse.jetty.server.Server">

  <Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
        <Arg>
          <New class="org.eclipse.jetty.http.ssl.SslContextFactory">
            <Set name="keyStore"><SystemProperty
name="lucidworksConfHome" />/keystore</Set>
            <Set name="keyStorePassword">secret</Set>
            <Set name="keyManagerPassword">secret</Set>
            <Set name="trustStore"><SystemProperty
name="lucidworksConfHome" />/truststore</Set>
            <Set name="trustStorePassword">secret2</Set>
            <Set name="needClientAuth">>false</Set>
          </New>
        </Arg>
        <Set name="port"><SystemProperty name="jetty.port" default="8888" /></Set>
        <Set name="maxIdleTime">30000</Set>
      </New>
    </Arg>
  </Call>

</Configure>

```

The keyStore and trustStore files must be located in the locations specified so they can be found on Jetty startup. They can be located anywhere on the server, as long as the correct locations are defined in the jetty-ssl.xml file.

Step 4: Modify jetty.xml for LWE-UI Component

The jetty.xml file found in \$LWS_HOME/conf/jetty/lwe-ui/etc needs to be modified to comment out the non-SSL connector. In the file, find the following section and add comment markers at the beginning and at the end (<!-- and -->, respectively) so it looks like this:

```

<!--
  <Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.bio.SocketConnector">
        <Set name="port"><SystemProperty name="jetty.port" default="8989" /></Set>
        <Set name="maxIdleTime">50000</Set>
        <Set name="lowResourceMaxIdleTime">1500</Set>
      </New>
    </Arg>
  </Call>
-->

```

Step 5: Modify jetty-ssl.xml for LWE-UI Component

In the directory `$LWS_HOME/conf/jetty/lwe-ui/etc` the file `jetty-ssl.xml` should be edited to activate the sample configuration. The configuration is currently commented out, but the comment tags should be removed and the `keyStore`, `keyStorePassword`, `keyManagerPassword`, `trustStore` and `trustStorePassword` parameters should be configured.

```
<Configure id="Server" class="org.eclipse.jetty.server.Server">

  <Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
        <Arg>
          <New class="org.eclipse.jetty.http.ssl.SslContextFactory">
            <Set name="keyStore"><SystemProperty
name="lucidworksConfHome"/>/keystore</Set>
            <Set name="keyStorePassword">secret</Set>
            <Set name="keyManagerPassword">secret</Set>
            <Set name="trustStore"><SystemProperty
name="lucidworksConfHome"/>/truststore</Set>
            <Set name="trustStorePassword">secret2</Set>
            <Set name="needClientAuth">>false</Set>
          </New>
        </Arg>
        <Set name="port"><SystemProperty name="jetty.port" default="8989"/></Set>
        <Set name="maxIdleTime">30000</Set>
      </New>
    </Arg>
  </Call>

</Configure>
```

The `keyStore` and `trustStore` files must be located in the locations specified so they can be found on Jetty startup. They can be located anywhere on the server, as long as the correct locations are defined in the `jetty-ssl.xml` file.

Step 6: Modify `jetty.xml` for the LWE-Connectors Component

The `jetty.xml` file found in `$LWS_HOME/conf/jetty/connectors/etc` needs to be modified to comment out the non-SSL connector and activate the SSL-connector. Unlike the LWE-Core and LWE-UI components, the Connectors component only requires modifying a single file.

In the file, find the following section and add comment markers at the beginning and at the end (`<!--` and `-->`, respectively) so it looks like this:


```
<--
  <Call name="addConnector">
    <Arg>
      <New class="org.eclipse.jetty.server.bio.SocketConnector">
        <Set name="port"><SystemProperty name="jetty.port" default="8765"/></Set>
        <Set name="maxIdleTime">50000</Set>
        <Set name="lowResourceMaxIdleTime">1500</Set>
      </New>
    </Arg>
  </Call>
-->
```

In the same file, uncomment the section "To add a HTTPS SSL Listener" to activate the sample configuration. After the comment tags are removed, configure the `keyStore`, `keyStorePassword`, `keyManagerPassword`, `trustStore` and `trustStorePassword` parameters.

```
<Call name="addConnector">
  <Arg>
    <New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
      <Arg>
        <New class="org.eclipse.jetty.http.ssl.SslContextFactory">
          <Set name="keyStore"><SystemProperty
name="lucidworksConfHome" />/keystore</Set>
          <Set name="keyStorePassword">secret</Set>
          <Set name="keyManagerPassword">secret</Set>
          <Set name="trustStore"><SystemProperty
name="lucidworksConfHome" />/truststore</Set>
          <Set name="trustStorePassword">secret2</Set>
          <Set name="needClientAuth">>false</Set>
        </New>
      </Arg>
      <Set name="port"><SystemProperty name="jetty.port" default="8765"/></Set>
      <Set name="maxIdleTime">30000</Set>
    </New>
  </Arg>
</Call>
```

The `keyStore` and `trustStore` files must be located in the locations specified so they can be found on Jetty startup. They can be located anywhere on the server, as long as the correct locations are defined in the file.

Step 7: Restart LucidWorks

After verifying that the `keyStore` and `trustStore` files are in the locations specified in each file, LucidWorks Search **must be restarted** for the changes to take effect.

Certificate Management

LucidWorks uses standard java jks format in keystores and truststores. Those stores can be managed using the standard Java [keytool](#).

Currently all certificates are managed outside of LucidWorks. There are no certificate management tools or admin displays for configuring SSL certificate related settings. All configuration tasks need to be made manually after installing LucidWorks and potentially repeated on all nodes where LucidWorks is running.

Client Certificates for LWE-Core and Connectors

It is possible to configure the LWE-Core and Connectors components to use certificates while communicating.

Prior to LucidWorks v2.5.2, the SSL Configuration API was used to define client certificates. This is now configured in `master.conf` as Java SSL system properties. To use these properties, open `master.conf` (found in `$LWS_HOME/conf` and edit these properties:

```
-Djavax.net.ssl.keyStore=conf/keystore.client  
-Djavax.net.ssl.keyStorePassword=secret2  
-Djavax.net.ssl.trustStore=conf/truststore.client  
-Djavax.net.ssl.trustStorePassword=secret3
```

The paths to the `keyStore` and `trustStore` should be entered as complete paths, or relative to `$LWS_HOME/app/bin`.



It is not possible to configure the LWE-UI component in this way

Configuring Mutually Authenticated SSL

LucidWorks supports securing communications to the core APIs with Mutual SSL authentication. This means the REST API and Solr API can be protected so that only clients that you trust can access these APIs. The system can also use mutually authenticated SSL internally to communicate to each Solr node when using distributed search.

The LucidWorks portions of SSL functionality can be configured by using the SSL Configuration API.

When configuring LucidWorks to use mutually authenticated SSL the container must also be configured to require certificates for authentication. In Jetty this is done by using `<set name="needClientAuth">true</Set>` in the related SSL Connector section of the Jetty configuration files (see above).



Mutual authentication is not supported for the LWE-UI component, and thus the Admin UI.

Debugging SSL Configuration

Reviewing logging events from the [LucidWorks log files](#) (either `core.YYYY_MM_DD.log` or `ui.YYYY_MM_DD.log`) may provide some hints about what is going on if SSL is not working as expected.

Common SSL Problems

Symptom: `javax.net.ssl.SSLHandshakeException: null cert chain`

Cause: Client is not sending client certificate. Reconfigure client so that it sends a client certificate with the request.

Symptom: `javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?`

Cause: Client is connecting to SSL endpoint without using SSL.



The cURL command line tool can be used to verify the SSL configuration. For example,:

```
curl --cacert <ca.crt> --key <host.key> --cert <client.crt>  
https://localhost:8443/dashboard
```

The link in this example is to the main LucidWorks Admin UI dashboard. Since this requires authentication, you should see the HTML indicating you will be redirected to the login page. If that's what you see, then SSL is properly set up.

Related Topics

- [Jetty doc on configuring SSL](#)
- [Java keytool](#)

Restricting Access to Content

LucidWorks Search provides three ways to restrict access to content through based on user identity:

- [Search Filters](#)
- [Access Control Lists](#)
- [Document-based Authorization](#)



Information for LucidWorks Search in the Cloud Users

Some sections following refer to editing the `solrconfig.xml` file, which is not possible for LucidWorks Search customers hosted in AWS or Azure.

Search Filters

[Search filters](#) provide the ability to limit the visibility of content only to specific users or user groups. For example, users in the finance role might be limited only to documents that satisfy the query `department:finance`. The LucidWorks Search Admin UI allows the creation of search filters that can be appended to all user queries. Usernames (manually created or supplied by the LDAP system) and/or groups (supplied by the LDAP system) can be mapped to search filters with the Search Filters page. You can also configure manual or LDAP search filters using the Roles API.

By default, LucidWorks comes configured with a default filter called "DEFAULT" that allows users to see all results for any query. This filter is defined in `solrconfig.xml`, and could be modified if needed:

```
<searchComponent class="com.lucid.handler.RoleBasedFilterComponent"
name="filterbyrole">
  <!-- Solr filter query that will be applied for users without group/role info -->
  <str name="default.filter">-*:*</str>
  <!-- Solr filter queries for roles, one role may have multiple filter queries.
  name is the role, value is the part of the filterquery that is to be formed. -->
  <lst name="filters">
    <str name="DEFAULT">*:*</str>
  </lst>
</searchComponent>
```

Note that this has defined that the default filter is `-*:*`. What this means is that someone without the DEFAULT role should see no results. However, since queries in LucidWorks Search are handled by the `/lucid` request handler, we have configured that handler to process searches for users without a role as though they had the DEFAULT role. This is in a later section of `solrconfig.xml`, where defaults are defined for the `/lucid` request handler (the below is truncated):

```
<lst name="defaults">
...
    <str name="role">DEFAULT</str>
...
</lst>
```

Access Control Lists

LucidWorks also supports [access control lists](#) (ACL) on Windows Share (SMB) and SharePoint data sources. ACL uses Windows Active Directory to control document access on a per-user basis. ACL filtering is configured for each data source, allowing you to have different authorizations depending on the definitions in each repository. To use this functionality, set up a Windows Share or SharePoint data source and configure the requisite fields.

If you do not need to configure ACL filtering on a per-data source basis, you can use the Filtering API to configure a Search Handler to perform the same functionality. Note that this is only supported for a Windows Share data source type. The Filtering API will configure the search handler in `solrconfig.xml` like this:

```
<searchComponent class="com.lucid.security.AclBasedFilterComponent" name="adfiltering">
  <str name="provider.class">com.lucid.security.ad.ADAclTagProvider</str>
  <str name="filterer.class">com.lucid.security.WindowsACLQueryFilterer</str>
  <lst name="provider.config">
    <str name="java.naming.provider.url">ldap://127.0.0.1</str>
    <str name="java.naming.security.principal">admin</str>
    <str name="java.naming.security.credentials">admin</str>
  </lst>
  <lst name="filterer.config">
    <str name="should_clause">*: * -data_source_type:smb</str>
  </lst>
</searchComponent>
```

In certain circumstances, you may need to add a `userFilter` or `groupFilter` parameter to the search component to properly implement your ACL filter.

Once created, the search component must be added to the `/lucid` request handler with the Search Components API.

Document-based Authorization

An application can enforce document visibility controls in front of LucidWorks simply by adding fields to each document that represent usernames, group membership, or other types of flags that help match a user with the content they are allowed to see in results. Generally these types of fields would be of type "string", possibly multi-valued. This technique is best suited to content extracted from a database or custom data source. The file and web crawling capabilities in LucidWorks do not index any security related attributes (though the file path itself may be useful for application-level restrictions).

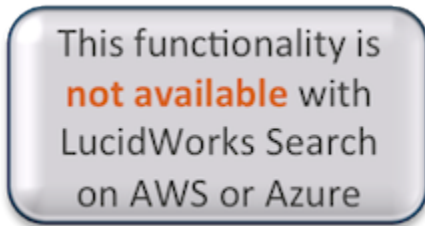
For example, documents could be indexed with an "owner" field. Here's a Solr XML file for this example:

```
<add>
  <doc>
    <field name="id">1</field>
    <field name="text">Bob's Document - For his eyes only\!</field>
    <field name="owner">bob</field>
  </doc>
  <doc>
    <field name="id">2</field>
    <field name="text">Jill's Document - Only she should find this</field>
    <field name="owner">jill</field>
  </doc>
</add>
```

Related Topics

- Windows Shares Data Sources
- SharePoint Data Sources
- Filtering Results

LDAP Integration



LucidWorks Search supports integrating user authentication with an existing LDAP system.

Two LDAP features are currently supported:

1. Authentication and Authorization of users (prerequisite for any other LDAP functionality)
2. User-to-group mapping (optional)

LDAP and built-in (API-based) user authentication are mutually exclusive. If LDAP is enabled, built-in authentication is not, and the reverse.

For standard LDAP integration, the LDAP administrative user only needs permissions to query the LDAP server for users and groups. We recommend that you create an LDAP admin user with only the necessary minimal user and group querying permissions for use with LucidWorks.

LucidWorks also allows you to authenticate users without LDAP administrative credentials. This method is called "queryless" authentication, because LucidWorks does not query the LDAP directory for user information. Rather, LucidWorks uses the attribute value plus the user's login and the base suffix as the user's DN. This method only works if the exact location of your LDAP user data is known and is the same for all relevant users. Another limitation of queryless authentication is that LucidWorks cannot find members of a group, only individual users.

It is also possible, using standard Java SSL functionality, to use certificate authentication with a SSL-enabled LDAP server. More information on that is available here:

<http://docs.oracle.com/javase/jndi/tutorial/ldap/security/ssl.html>.

For information about filtering search results based on LDAP permissions, see [Restricting Access to Content](#) and [Search Filters](#).

Enabling LDAP

These steps need to be completed to successfully enable LDAP. Each step is required and should be done in this order:

1. Configure the [LDAP Configuration File](#) with the instructions below.

2. Map at least one user to have admin permissions using the LDAP section of the [Settings page](#). Because the built-in authentication is disabled when LDAP authentication is enabled, you cannot map a user or group to the Admin authorization after LDAP is enabled. If no one has Admin authorization, no one will be able to access the Administration User Interface. So, before enabling LDAP, go to the [System Settings](#) page and map an LDAP username or a group to "Admin UI" by adding it to the Group or User section of the Admin UI definition.
3. Enable LDAP by setting the environment variable `lweui.ldap.enabled` to **true** in the `master.conf` file found in `$LWS_HOME/conf/`.
4. [Restart LucidWorks](#).

LDAP Configuration File

The main configuration file for configuring LDAP is `ldap.yml`, found in the `$LWS_HOME/conf/` directory. The default settings must be modified as needed for LucidWorks to connect to the LDAP server and query the database for user authentication. If LDAP is already enabled and this file is edited, you will need to [restart the server](#) for changes to take effect.

Below is the main section of the `ldap.yml` configuration file that needs to be edited. Note that the file also includes sample configurations for standard LDAP authentication, queryless authentication, and Microsoft ActiveDirectory integration for use with [Windows Shares data sources](#).




Lines Must Be Indented

When customizing the `ldap.yml` file, keep in mind that the attributes must be indented at least two spaces. So, when removing the hash mark (`#`), do not remove the extra spaces. All lines must also be indented the same number of spaces (so, if some lines are indented three spaces, then all lines must be indented three spaces).


```
#####
# Warning: Always restart the application after adjusting
#   your LDAP config, or unpredictable behavior may result.
#####
# production:
#   host: localhost
#   port: 389 # 636 for SSL
#   attribute: uid
#   base: dc=xyz,dc=corp,dc=com
#   # user_query: '$ATTR=$LOGIN' # default query is '$ATTR=$LOGIN', set this if you
#   need something more complex
#   admin_user: cn=Manager,dc=xyz,dc=corp,dc=com # If you don't have an admin
#   password, you can disable
#   admin_password: secret # admin login in the UI "Settings"
#   page
#   ssl: false
#   group_base: ou=groups,dc=xyz,dc=corp,dc=com
#   group_membership_attribute: uniqueMember
#   group_name_attribute: cn
#   # group_query: '(&(objectclass=groupOfUniqueNames)($ATTR=$USER))' # default query
#   is '$ATTR=$USER' where $USER is user's DN
```

The attribute definitions included in the `ldap.yml` file are as follows:

Attribute	Definition
host	The hostname of the LDAP server that contains the user information.
port	The port to use while connecting to the LDAP server that contains the user information.
attribute	The attribute of the user object that the system will use to search for the user, or assume when constructing an explicit DN via query-less authentication.
base	Search base for user queries, or suffix appended to attribute + login for queryless authentication.
user_query	Optional: supplies an arbitrarily complex query if the default user query is not sufficient. Variable substitutions are as follows: \$ATTR will be substituted with the value of 'attribute' from above; \$LOGIN will be substituted with the value the user entered in the login form in the UI. Search is performed using 'base' as a search base.
admin_user	Administrative login to use for searching the directory. Not used for queryless authentication.

admin_password	Administrative password to use for searching the directory. Not used for queryless authentication.
ssl	Enable/disable SSL.
group_base	Search base for group queries. Not used with queryless authentication.
group_membership_attribute	The attribute to look for in the group object that will contain members' user DNs.
group_name_attribute	The attribute of the group object that the system will use to search for the group.
group_query	<p>Optional: supplies an arbitrarily complex query if the default group query is not sufficient. Variable substitutions are as follows: <code>\$ATTR</code> will be substituted with the value of 'group_name_attribute'; <code>\$USER</code> will be substituted with the logged-in user's fully-qualified LDAP DN. Search is performed using 'group_base' as a search base.</p> <div data-bbox="583 932 1404 1169"> <p> The default query (<code>\$ATTR=\$USER</code>) does not specify the object type for groups. Several different group object types are common, such as group, groupOfNames, groupOfUniqueNames, and so on. Therefore, non-group objects may also match if they contain a matching attribute.</p> </div>

User to Group Mappings

LucidWorks supports mapping users to groups with the `group_membership_attribute` setting. This allows LucidWorks to do an additional query while the user is logging in to find all the groups the user is a member of.

Manual User Management

LucidWorks also includes a REST API that allows creation and authentication of users. Using this API and the Perl Examples provided with the application, users can be created, passwords reset, and accounts deleted. As mentioned previously, API-based user management and LDAP authentication are mutually exclusive: you can only use one user management method.

Related Topics

- [Restricting Access to Content](#)
- [Search Filters](#)

Solr Direct Access

LucidWorks Search is Solr-powered at its core. Solr, an Apache Software Foundation project, provides an easy-to-use HTTP interface above and beyond Lucene, a very fast and scalable Java search engine library. Both Solr and Lucene are entirely open source, available under the Apache Software License.

LucidWorks Search exposes the Solr interface directly. This means that applications can leverage both Solr's power and openness and LucidWorks Search's ease of use.

This guide covers Solr when LucidWorks and Solr intersect but it does not provide an extensive overview of the inner workings of Solr, and in places assumes some basic knowledge of Solr. For a good introduction to Solr, the Lucene/Solr community has produced an [Apache Solr Reference Guide](#) which provides a lot of information about how Solr works "under the hood".

Solr Version

For information about the Solr version included in this release of LucidWorks, see the `SOLR_VERSION.txt` file in `$LWS_HOME/app/SOLR_VERSION.txt`. For LucidWorks v2.6.3, we have included Solr version 4.6 (the official release).

You can also get detailed Solr version information for all releases of LucidWorks Search from our public Github fork here: <https://github.com/lucidimagination/lucene-solr>. To see information for a specific LucidWorks version, select the tag for that version from the "Switch Tags" drop-down list. Please note, however, that this is not a stand-alone, runnable Lucene or Solr release; it is intended as a source reference only.

How the LucidWorks-Bundled Solr is Different

The primary difference between using Solr and LucidWorks is the base URL. Solr's example application is accessed by default at `http://localhost:8983/solr/`, whereas the LucidWorks default collection instance of Solr is rooted at `http://localhost:8888/solr/collection1/`. If using multiple collections, replace `collection1` with the correct collection name. The Solr URL for each collection is displayed under each collection listing on the main [Collections](#) page in the Admin UI.

In addition, some of the examples that are usually included with Solr are not included with LucidWorks. This includes detailed examples and explanations that are provided in the `schema.xml` and `solrconfig.xml` files. Those examples will likely still work with LucidWorks, but would need to be inserted manually into those files.

Other differences are mentioned specifically in sections that discuss certain features. If a limitation with Solr is not mentioned, it can be assumed that the Solr functionality works as you would expect with a stand-alone Solr instance.

Adding Solr Plugins

Generally speaking, most plugins to Solr should work with LucidWorks Search, provided that they are compatible with the Solr version used with LucidWorks Search (see [Solr Version](#) above). As described in the [Solr Wiki page on Solr Plugins](#), there are two options for integrating plugins:

1. "Place your JARs in a `lib` directory in the `instanceDir` of your `SolrCore`." For LucidWorks Search, this would mean the `lib` directory of your collection `instance_dir`. For example, if you wanted to use the plugin with the default collection, `collection1`, you would put the relevant JARs in `$LWS_HOME/conf/solr/cores/collection1_0/bin`. You can find the `instance_dir` name with the Collections API. The name indicates a directory name, always relative to `$LWS_HOME/conf/solr/cores`.
2. "Use the `lib` directive in your `solrconfig.xml` file to specify an arbitrary JAR path, directory of JAR files, or a directory plus regex that JAR file names must match." This alternative allows you define the path in `solrconfig.xml` for your collection using the `<lib>` directive. More information on using this directive is available in the Apache Solr Reference Guide section on [Lib Directives in SolrConfig](#).

Either of these approaches will allow integration of a Solr-based plugin with LucidWorks Search. If the plugin will be used with multiple LucidWorks Search collections, pick either approach here and configure the use of the plugin for one collection. Once you've verified that it works successfully with LucidWorks Search, you can use that single collection to create a [Collection Template](#) for use as the basis for future collections.

If there is configuration to be done in `solrconfig.xml` or `schema.xml` (or other configuration files) in order to properly use the plugin, you will need to make those changes as a separate step and by manually editing the files. If the changes conflict with or modify the LucidWorks Search defaults, the Admin or Search UI may behave abnormally. It's best to do thorough testing before moving to production with any plugin.

More information on how to create a custom plugin is available from the Solr Wiki at <http://wiki.apache.org/solr/SolrPlugins>.

Related Topics

- [Apache Solr Reference Guide](#)
- [Apache Solr project homepage](#)
- [Apache Solr Wiki](#)
- [Solr Plugins](#) from the Solr Wiki

Performance Tips

A number of configuration items can be manipulated for better performance when benchmarking LucidWorks. Implementing some of these optimizations may require directly configuring Solr via `schema.xml` and `solrconfig.xml`. See the [Apache Solr Reference Guide](#) for details on Solr customizations that may be right for your implementation.

- Ensure that you are running the JVM in server mode.
- Allocate only as much memory as needed to the JVM heap. The rest should be left free to allow the operating system to cache as much of the Lucene index files as possible.

Improving indexing speed

- Minimize indexing the same content in more than one field. Each field should be either indexed on its own or Solr's [copyField](#) functionality can be used to copy it to an indexed catch-all field.
- Avoid storing the same content more than once. The target field of copyField commands should almost never be stored.
- Avoid commits during the indexing process. Turn off Solr auto-commit and avoid explicitly committing until indexing has completed.
- Disable rules processing if not using business rules as part of your implementation. See the section on [Disabling Business Rules](#) for details on how to disable rules processing.

Improving Search speed

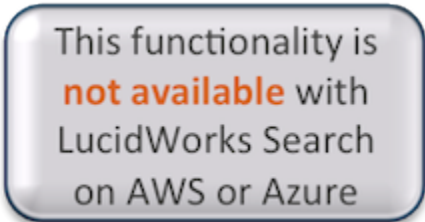
- Perform a variety of searches before starting any timings. This warms up the server JVM, and causes parts of the index, commonly used sort fields and filters to be cached by the operating system.
- Search in as few fields as possible. A single indexed catch-all text field containing the contents of all the other searchable fields (generated by **copyField** commands) will be faster to search than a multi-field query across many indexed fields.
- If necessary, turn off relevancy enhancers such as proximity phrase queries, date recency boosts, and synonym expansion to generate benchmarks for comparison with later tests when those features are re-enabled.
- Retrieve the minimum number of [stored fields](#) that still provide a optimal search experience for users.
- Only retrieve the number of documents that are immediately necessary. The **start** and **rows** query arguments may be used to request pages of results.
- Disable rules processing if not using business rules as part of your implementation. See the section on [Disabling Business Rules](#) for details on how to disable rules processing.
- For a large index (on *NIX), force key parts of the indexed portion into operating system cache by changing to the index directory and executing `cat *.prx *.frq *.tis > /dev/null`

- Review the section on Wildcards at Start of Terms if leading wildcards have been enabled for important performance considerations.

Related Topics

- [Expanding Capacity](#)

Expanding Capacity



This functionality is
not available with
LucidWorks Search
on AWS or Azure

As your search application grows, you may need to scale the system to add space for indexes or to increase query responsiveness. This section discusses advanced deployment options to enhance system performance and ensure seamless application scaling.

With Solr 4, which is included with LucidWorks Search, the best way to scale is in SolrCloud mode. How to start LucidWorks in SolrCloud mode is discussed in the section [Using SolrCloud in LucidWorks](#).

If you only need to extend your index across multiple servers [Index Replication](#) shows how to configure multiple shards for a master-slave environment. Or you can use [Distributed Search and Indexing](#) to distribute search and indexing processes across multiple servers or shards for peak performance. Note, however, that distributed search and replication are no longer in active development by the Solr community.

Using SolrCloud in LucidWorks

SolrCloud is a set of Solr features that expands the capabilities of Solr's distributed search, simplifying the creation and management of Solr clusters. SolrCloud is still under active development, but already supports the following features:

- Central configuration for the entire cluster
- Automatic load balancing and fail-over for queries
- Zookeeper integration for cluster coordination and configuration

For an introduction to SolrCloud, and how it is different from index replication, see the LucidWorks Knowledgebase article [What is SolrCloud?](#). In addition, the Apache Solr Reference Guide includes an extensive [section on SolrCloud](#), which includes background information and configuration instructions. Some changes have been made for LucidWorks Search, however, which are described below.

LucidWorks Search implements SolrCloud as a **purely Solr feature**; to manage SolrCloud shards and replicas, you should refer to and use instructions designed for a purely Solr installation. There are only a few caveats and modifications for LucidWorks Search, detailed below, specifically for bootstrapping ZooKeeper and the cluster nodes.

Topics discussed in this section:

- [Enabling SolrCloud Mode](#)
 - [Using the Embedded ZooKeeper](#)
 - [Bootstrapping Solr vs. LucidWorks Search](#)
- [How SolrCloud Works with LucidWorks](#)
 - [Replicated Configurations](#)
 - [Using the Admin UI in SolrCloud Mode](#)
 - [Feature Limitations](#)
 - [Collections APIs](#)
- [Using a Stand-Alone ZooKeeper Instance or Ensemble](#)
- [Related Topics](#)

Enabling SolrCloud Mode

LucidWorks Search includes an installer that can install the application on each node of the planned SolrCloud cluster. For details on using this approach, see the section SolrCloud Cluster Installation. This approach will allow you to install three ZooKeeper instances to create a quorum, and then install as many LucidWorks Search nodes as needed.


The standard instructions for starting SolrCloud are modified slightly for LucidWorks Search. Commands within the installer take these modifications into account, but if starting without the installer, refer to the modifications described below.

While much of the SolrCloud documentation in the Apache Solr Reference Guide [section on SolrCloud](#) can be used, it is important to only start LucidWorks Search in SolrCloud mode with the instructions included here.

Using the Embedded ZooKeeper

It's possible to make two standalone, or single server, installations communicate with each other in SolrCloud mode using the ZooKeeper instance embedded with Lucidworks Search. This can be useful to create a simple two-node cluster when just starting to learn how this functionality can work for your search application. With this approach, two separate installations are made (as described in the section Single Server Installation). Then one installation is started with commands to bootstrap configurations and start ZooKeeper.

Because we need two servers for this example, we will make two installations of LucidWorks, one on the server "example" and the other on the server "example2". During installation, do not start LucidWorks Search. Instead, start the two installations manually, as shown below.

 We recommend that you only install LucidWorks using the installer application; copying the LucidWorksSearch directory to another directory to create another server may cause conflicts with ports. Information on installing LucidWorks is available in the section on Installation.

The installation in `example` should use port 8983 for the LWE-Core component, which will be changed from the default during the installation process. The installation on `example2` should use the default port (8888) for the LWE-Core component. If enabling other components, be sure to modify the ports for each installation as well. If new to LucidWorks, see the section on [Working With LucidWorks Search Components](#) for more information about the components. Your port selections might look like this:

Component	example Ports	example2 Ports
LWE-Core	8983	8888
LWE-Connectors	8965	8765
LWE-UI	8889	8989

ZooKeeper will run on the LWE-Core port + 1000, so in this scenario we expect ZooKeeper to run on port 9983. It's important to keep that in mind while planning the installation ports so there isn't an inadvertent conflict with LucidWorks Search ports.

- ✔ SolrCloud uses ZooKeeper to manage nodes, and it's worth taking a look at the [ZooKeeper website](#) to understand how ZooKeeper works before configuring SolrCloud. Solr can embed ZooKeeper, but for a production use, it's recommended to run a ZooKeeper ensemble, as described in the [ZooKeeper section](#) of the SolrCloud wiki page.

Starting LucidWorks Search

To start LucidWorks Search in SolrCloud mode, use the usual LucidWorks start script, but pass some Java options to it. To start `example`, you would use a command like this:

```
Start 'example'
```

```
$LWS_HOME/app/bin/start.sh -lwe_core_java_opts "-Dbootstrap_conf=true -DzkRun  
-DnumShards=2"
```

The `bootstrap_conf` allows copying of the configuration files for each collection to the nodes, while `zkRun` starts ZooKeeper. The `numShards` value defines how many nodes there will be in the cluster. Be sure to set this accurately, as Solr cannot yet easily increase the number of shards without re-bootstrapping the cluster.


We only need to pass `bootstrap_conf` and `numShards` the first time LucidWorks is started in SolrCloud mode. In subsequent LucidWorks restarts, start this leader node with `./start.sh -lwe_core_java_opts "-DzkRun"`. The `-DzkRun` could be added to `master.conf`, in which case the `start.sh` script alone would start ZooKeeper each time.

To start the next nodes of the cluster, we still use the start script, but with some different options. This would start `example2`:

```
Start 'example2'
```

```
$LWS_HOME/app/bin/start.sh -lwe_core_java_opts "-DzkHost=localhost:9983"
```

Note that the port defined as the `zkHost` is the port of the LWE-Core component + 1000. So, if LWE-Core on our `example` server was defined at port 8983, ZooKeeper would be started at port 9983.

-  The above instructions assume a Linux-based operating system. For Windows-based systems, use `start.bat` as in these examples:

Start example:

```
$LWS_HOME\app\bin\start.bat -lwe_core_java_opts "-Dbootstrap_conf=true -DzkRun
-DnumShards=2"
```

Start example2:

```
$LWS_HOME\app\bin\start.bat -lwe_core_java_opts "-DzkHost=localhost:9983"
```

If you have used the installer to install LucidWorks in SolrCloud mode, the required commands have been added to the `master.conf` for each server, and no special start or stop instructions are required for restarts. In that case, you would not run the embedded ZooKeeper; instead you would have installed and configured a quorum, and the `zkHost` parameters have been added to the `master.conf` file.

Bootstrapping Solr vs. LucidWorks Search

This table outlines the differences between the Solr instructions for bootstrapping SolrCloud mode and the LucidWorks Search instructions. It is meant as a summary if you are already familiar with how SolrCloud works.

SolrCloud	LucidWorks Search
Use <code>start.jar</code>	Use <code>start.sh</code> or <code>start.bat</code> with <code>-lwe_core_java_opts</code> defined
Use <code>bootstrap_confdir</code> to upload configuration files to ZooKeeper	<code>bootstrap_conf=true</code>
Use <code>collection.configName</code>	Not needed with <code>bootstrap_conf=true</code>
Default configuration directory is <code>./solr/collection1/conf</code>	Default configuration directory is <code>\$LWS_HOME/conf/solr/cores/collection1_0/conf</code>

How SolrCloud Works with LucidWorks

There are some caveats to using SolrCloud with LucidWorks Search, as it is so far only partially integrated with the system. Future releases of LucidWorks Search will contain more tight integration points with SolrCloud functionality.

Replicated Configurations

When running LucidWorks Search in SolrCloud mode, some LucidWorks Search-specific features are not yet fault tolerant and highly available. While the index and configuration files are fully SolrCloud supported, the following are not currently replicated across shards:

- Data sources and their related metadata (such as crawl history)
- The LucidWorks user database, which stores manually created users (such as the default "admin" user)
- User alerts
- LDAP configuration files
- SSL configuration

Even though these features aren't replicated, they can still be used with LucidWorks Search in SolrCloud mode. The files that hold this metadata are in the `$LWS_HOME/conf` folder and could be copied to the other nodes in the cluster to act as backup if the main node goes down for any length of time. This is a manual process and not yet automated by LucidWorks Search.

Using the Admin UI in SolrCloud Mode

To accommodate for the lack of replicated configurations, we recommend that you do a full LucidWorks Search installation (i.e., [all components](#)) on every machine in your cluster. You should then choose one node to use for the Admin UI. This is the node that will store your data sources and associated metadata. Another node can be chosen as the node that does crawling, or you can use the same node used by the Admin UI. Document updates will still be sent to the nodes, via the index update processes that make up SolrCloud functionality.

If the node used for the Admin UI goes down, you can choose another node to act as the Admin UI node, but unless the related configuration files have been copied to that node you will not have the same user accounts and data sources in the other nodes. Once you bring the node originally used for the Admin UI back, it should still have your data sources and other LucidWorks-specific metadata.

You can configure LucidWorks Search to not start the Admin UI by changing `$LWS_HOME/conf/master.conf` and setting the `lweui.enabled` parameter to 'false'.

Feature Limitations

The following LucidWorks features may encounter significant problems when working in SolrCloud mode:

- Click Scoring cannot be used in SolrCloud mode at this time.
- Auto-complete-related suggestions should be pulled from a single index node if auto-complete is enabled by adding `&distributed=false` to any query. Distributed auto-complete indexing is possible but requires configuration of the auto-complete indexing on each node and adding a 'query' component to the autocomplete requestHandler in `solrconfig.xml`.
- De-duplication does not work in SolrCloud due to a bug in Solr ([SOLR-3473](#)).
- SSL does not work with SolrCloud due to a bug in Solr ([SOLR-3854](#)).

- Log indexing and query statistics in the Admin UI will be inconsistent. If you are using LucidWorks Search in SolrCloud mode or with each component installed on a different server, please see the section Log Indexing with Separated Components for details on how to make sure your logs are fully indexed.

Collections APIs

LucidWorks Search and Solr both have Collections APIs. They are not duplicates, even though they share the same parameters. It is important, however, to only use the LucidWorks Search Collections API to create collections, because of the issues described in the section [Replicated Configurations](#). The LucidWorks Search Admin UI also uses the LucidWorks Collections API to create collections.

When creating a new collection (with either the Admin UI or the API), and you are working in SolrCloud mode, you can specify the number of shards to break it up into. This number, however, cannot be higher than the number of shards defined when LucidWorks Search was bootstrapped.

Behind the scenes, the LucidWorks Search Collections API update LucidWorks Search-specific collection configuration files and also uses Solr's Collection API to create the collection in Solr. This has some ramifications for LucidWorks Search:

- Solr's Collection API does not allow defining the instanceDir or the dataDir, so there is no way for LucidWorks Search to instruct Solr to create the new collection directories in the same place on the filesystem as the pre-existing collections that ship with LucidWorks Search. Solr creates collections by default with the `conf` and `data` directories in the same location, but the LucidWorks Search directory structure separates those directories to `$LWS_HOME/conf/solr/cores` and `$LWS_HOME/data/solr/cores`. Because Solr's Collection API does not allow setting the path values explicitly, they are created in Solr's default location. What this means is that new collections created in SolrCloud mode will be located in a different location from the pre-existing collections (i.e., they will be located under `$LWS_HOME/conf/solr` and the data directory will not be located under `$LWS_HOME/data/solr`). This is normal and will not have any impact on document indexing or searching.
- Solr's Collection API itself uses Solr's CoreAdmin API to asynchronously create cores on each node. For this reason, the collection will appear to be renamed as `<collection>_shard<x>_replica<y>`. LucidWorks Search will mostly display the correct name, but the directory on the server will show the core name (and each core on each node will be named differently). The Solr Admin UI will also display the core name in the Core dropdown list. If you are accessing the Solr Admin for several different nodes, this may cause some initial confusion. Essentially, LucidWorks displays information about a collection, but Solr displays information about the specific core you are looking at. For more information about the differences between cores and collections in Solr, also refer to the [SolrCloud Glossary](#) and other pages on SolrCloud in the Apache Solr Reference Guide.

Using a Stand-Alone ZooKeeper Instance or Ensemble

If you review the Solr Reference Guide or any of the Solr documentation about SolrCloud, you may notice that using the Apache ZooKeeper instance that is included with Solr is not recommended for real production systems. This is because the embedded Zookeeper will not provide sufficient failover; the ZooKeeper instance is dependent on the Solr instance so if one of the Solr instances is shut down, an associated ZooKeeper instance will also be shut down.

For this reason, the LucidWorks installer includes the ability to install a ZooKeeper quorum while installing LucidWorks Search.

If you have an existing ZooKeeper, or an existing SolrCloud setup, the Apache Solr Reference Guide provides information about how to use a stand-alone ZooKeeper instance at [Setting Up an External ZooKeeper Ensemble](#). That information is worth reviewing before installing a stand-alone ZooKeeper. The same instructions apply if used with LucidWorks Search, with the exception of the bootstrapping instructions as described in the earlier section [Starting LucidWorks Search](#) (above).



When using stand-alone ZooKeeper with LucidWorks Search, you need to take care to keep your version of ZooKeeper updated with the latest version distributed with Solr and LucidWorks Search. Since you are using it as a stand-alone application, it does not get upgraded when you upgrade LucidWorks Search.

Solr 4.0 and LucidWorks 2.5.0 and 2.5.1 use Apache ZooKeeper v3.3.6.

Solr 4.1 and higher, and LucidWorks 2.5.2 and higher, use Apache ZooKeeper v3.4.5.

Related Topics

- [Getting Started with SolrCloud](#) from the Apache Solr Reference Guide
- [SolrCloud Wiki page](#)

Index Replication

This functionality is
not available with
LucidWorks Search
on AWS or Azure

⚠ As of Solr 4.0, SolrCloud is the preferred way to distribute indexes for redundancy, failover, and improved performance. Index Replication and Distributed Search are considered obsolete technologies; while still supported, they are not in active development. See the section on [Using SolrCloud in LucidWorks](#) for more information on using SolrCloud with LucidWorks Search.

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 master server's index is replicated on the slaves, which then process requests such as queries.

LucidWorks Search supports index replication, but it is not configured through the Admin UI. Instead, replication configuration requires editing XML configuration files in the Solr release included with LucidWorks Search. This section explains how replication works and how to edit the configuration files. Detailed examples are provided, so even if you're new to XML and Solr configuration, you should be able to set up and configure master/slave replication servers with ease.

✔ When the [Click Scoring Relevance Framework](#) is enabled, LucidWorks ensures that also the click boost data is replicated together with index files. See the section on [Click Scoring Tools and Index Replication](#) for more information.

Configuring Replication on the Master Server

To set up replication, you will need to edit the `solrconfig.xml` file on the master server. To edit the file, you can use an XML editor or even a simpler tool such as Notepad on a PC or TextEdit on a Mac.

Within the `solrconfig.xml` file, you will edit the definition for a Request Handler. A Request Handler is a Solr process that responds to requests. In this case, you will be configuring the Replication RequestHandler, which processes requests specific to replication.

The example below shows how to configure the Replication RequestHandler on a master server.

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <!-- Replicate on 'optimize'. Other values can be 'commit', 'startup'.
         It is possible to have multiple entries of this config string -->
    <str name="replicateAfter">optimize</str>
    <!-- Create a backup after 'optimize'. Other values can be 'commit', 'startup'.
         It is possible to have multiple entries of this config string.
         Note that this is just for backup, replication does not require this.
    -->
    <!-- <str name="backupAfter">optimize</str> -->
    <!-- If configuration files need to be replicated give the names here,
         separated by comma -->
    <str name="confFiles">schema.xml,stopwords.txt,elevate.xml</str>
    <!-- The default value of reservation is 10 secs. See the documentation
         below. Normally, you should not need to specify this -->
    <str name="commitReserveDuration">00:00:10</str>
  </lst>
</requestHandler>
```

Operations that Trigger Replication

The value of the `replicateAfter` parameter in the ReplicationHandler configuration determines which types of events should trigger the creation of snapshots for use in replication.

The `replicateAfter` parameter can accept multiple arguments.

replicateAfter Setting	Description
startup	Triggers replication whenever the master index starts up.
commit	Triggers replication whenever a commit is performed on the master index.
optimize	Triggers replication whenever the master index is optimized.

If you are using `startup` setting for `replicateAfter`, you'll also need a `commit` or `optimize` if you want to trigger replication on future commits/optimizes as well. If only the `startup` option is given, replication will not be triggered on subsequent commits/optimizes after it is done for the first time at the start.

Configuring Replication on Slave Servers

The code below shows how to configure a ReplicationHandler on a slave server.


```

<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://master.solr.company.com:8983/solr/corename/replication</str>
    <!-- Interval in which the slave should poll master. Format is HH:mm:ss.
         If this is absent slave does not poll automatically.
         But a fetchindex can be triggered from the admin or the http API
    -->
    <str name="pollInterval">00:00:20</str>
    <!-- THE FOLLOWING PARAMETERS ARE USUALLY NOT REQUIRED -->
    <!-- To use compression while transferring the index files.
         The possible values are internal|external
         if the value is 'external' make sure that your master Solr
         has the settings to honor the accept-encoding header.
         see here for details http://wiki.apache.org/solr/SolrHttpCompression
         If it is 'internal' everything will be taken care of automatically.

         USE THIS ONLY IF YOUR BANDWIDTH IS LOW.
         THIS CAN ACTUALLY SLOW DOWN 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>

```

The master server is unaware of the slaves. Each slave server continuously polls 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:

1. The slave issues a `filelist` command to get the list of the files. This command returns the names of the files as well as some metadata (e.g., size, a lastmodified timestamp, an alias if any).

2. 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.
3. 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 replication process will simply abort.
4. After the download completes, all the new files are 'mv'ed to the live index directory, and the file's timestamp is set to be identical to the file's counterpart on the master master.
5. A commit command is issued on the slave by the Slave's ReplicationHandler, and the new index is loaded.

Configuring Replication on a Repeater Server

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's an example of a ReplicationHandler configuration for a repeater:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">commit</str>
    <str name="confFiles">schema.xml,stopwords.txt,synonyms.txt</str>
  </lst>
  <lst name="slave">
    <str
name="masterUrl">http://master.solr.company.com:8983/solr/corename/replication</str>
    <str name="pollInterval">00:00:60</str>
  </lst>
</requestHandler>
```

Replicating Configuration Files

To replicate configuration files, list them with the `confFiles` parameter in the master's configuration. Only files found in the `conf` directory of the master's Solr instance will be replicated.

Solr replicates configuration files only when the index itself is replicated. 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.

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.

Unlike the index files, where the timestamp is good enough to figure out if they are identical, configuration files are compared against their checksum. If a replication involved downloading at least one configuration file with a modified checksum, the `ReplicationHandler` issues a `core-reload` command instead of a `commit` command.

Replicating the `solrconfig.xml` File

To keep the configuration of the master servers and slave servers in sync, you can configure the replication process to copy configuration files from the master server to the slave servers. In the `solrconfig.xml` on the master server, include a `confFiles` value 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 `:`.

Related Topics

- [Using SolrCloud in LucidWorks](#)
- [Scaling and Distribution chapter](#) from the Apache Solr Reference Guide

Distributed Search and Indexing

This functionality is
not available with
LucidWorks Search
on AWS or Azure

⚠ As of Solr 4.0, SolrCloud is the preferred way to distribute indexes for redundancy, failover, and improved performance. Index Replication and Distributed Search are considered obsolete technologies; while still supported, they are not in active development. See the section on [Using SolrCloud in LucidWorks](#) for more information on using SolrCloud with LucidWorks Search.

Consider using distributed search when an index becomes too large to fit on a single system, or when a single query takes too long to execute. Distributed search can reduce the latency of a query by splitting the index into multiple shards and querying across all shards in parallel, merging the results.

Distributed search should not be used if queries to a single index are fast enough but one simply wishes to expand the capacity (queries per second) of the system. In this case, standard [Index Replication](#) should be used.

Distributed Indexing

To utilize distributed search, the index must be split into shards across multiple servers. Each shard is a LucidWorks Search server containing a complete index that can be queried independently, but which only contains a fraction of the complete search collection.

- ❌ If using distributed indexing with a Solr XML data source type, you may encounter a situation where the crawl never ends without a restart of LucidWorks. This is due to a problem in the distributed index processor and the way Solr XML files are crawled by LucidWorks.

There are two possible solutions to this problem:

1. Use [SolrCloud](#). The distributed indexing is handled automatically by ZooKeeper, and provides automatic failover in case of server failure.
2. Disable the `DistributedUpdateProcessor` on all but the primary, master, node. It is not really required to be running on slave nodes since LucidWorks crawlers send their files through only one node during processing.

Manual Distributed Indexing

One method of splitting the search collection into multiple shards is to index some documents to each shard instead of sending all documents to a single shard. Updates to a document should always be sent to the same shard, and documents should not be duplicated on different shards.

Manual Configuration

A Distributed Update Processor can be enabled to automatically support distributed indexing by sending update requests to multiple servers (shards).

Enabling distributed indexing is done via the `solrconfig.xml` file, found in `$LWS_HOME/solr/cores/collection/conf` (replace `collection` with the name of the collection that is being configured for distributed indexing). By default it is not enabled. The `solrconfig.xml` file needs to be installed on each shard, and the shards should be listed in the same order in each file.

The distributed update processor is controlled by two parameters, `shards` and `self`, which may either be specified in `solrconfig.xml`, or supplied with a specific update request to Solr.

- `shards` lists the servers in the cluster. The list should be exactly the same (that is, in the same order) in the configuration file for every server in the cluster.
- `self` should be different for each server in the cluster and should match the entry in `shards` for the particular server. It is used to allow updates for the particular server to be directly added rather than going through the HTTP interface. If it is missing, distributed update will still work, but will be less efficient.

To start using distributed indexing, find the following section in `solrconfig.xml`, and uncomment the shard location definitions. Below is an example of shard definition that is not commented out.

```
<updateRequestProcessorChain name="lucid-update-chain">
  <processor class="com.lucid.update.DistributedUpdateProcessorFactory">
    <!-- example configuration...
    "shards should be in the *same* order for every server
    in a cluster. Only "self" should change to represent
    what server *this* is. -->

    <str name="self">localhost:8983/solr</str>
    <arr name="shards">
      <str>localhost:8983/solr</str>
      <str>localhost:7574/solr</str>
    </arr>
  </processor>
  <processor class="solr.LogUpdateProcessorFactory">
    <int name="maxNumToLog">10</int>
  </processor>
  <processor class="com.lucid.update.FieldMappingUpdateProcessorFactory"/>
  <processor class="solr.RunUpdateProcessorFactory"/>
</updateRequestProcessorChain>
```

Indexing Documents

If distributed indexing has been configured as above, then any indexing initiated from the LucidWorks Search administration user interface, such as crawling directories, will be appropriately handled by sending some documents to each server. One can use the distributed update processor in conjunction with any update handler while directly updating Solr. The `/update/xml` and `/update/csv` update handlers are already configured to use `distrib`, the distributed update processor, by default.

If an update handler has not been configured to use the distributed update processor, it may be specified in the URL via the `update.processor` parameter:

```
http://localhost:8888/solr/collection1/update?update.processor=distrib
```

If the `self` and `shards` parameters are not configured in `solrconfig.xml`, then they may be specified as arguments on the update url.

```
http://localhost:8888/solr/collection1/update?update.processor=distrib&self=localhost:8888
```

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.

Distributed Search

After a logical index is split across multiple shards, distributed search is used to make requests to all shards, merging the results to make it appear as if it came from a single server.

Programmatic Distributed Search

One can use distributed search with Solr request handlers such as `standard`, `dismax`, or `lucid` (the handler used by the LucidWorks Search), or any other search handler based on `org.apache.solr.handler.component.SearchHandler`.

Supported Components

The following Solr components currently support distributed searching:

- The Query component that returns documents matching a query
- The Facet component, for `facet.query` and `facet.field` requests where `facet.sorted=true` (the default: return the constraints with the highest counts)
- The Highlighting component, which highlights results
- The Debug component

The presence of the `shards` parameter in a request will cause that request to be distributed across all shards in the list. The syntax of `shards` is

`host1:port1/base_url1,host2:port2/base_url2,...`

The example below would query across 3 different shards, combining the results:

```
http://localhost:8888/solr/collection1/select?shards=localhost:8983/solr,localhost:7574/solr
```

As a convenience to clients, a new request handler could be created with `shards` set as a default like any other ordinary parameter.



The `shards` parameter should not be set as a default in the standard request handler as this could cause infinite recursion.

Scalability and Fault Tolerance

To provide fault tolerance and increased scalability, standard [replication](#) can be used to provide multiple identical copies of each index shard. Each shard would have a master and multiple slaves.

Indexing in a Fault Tolerant Distributed Configuration

Only the master for each shard should be configured in distributed indexing or specified to the distributed update processor. There is no fault tolerance while indexing - if the master for a shard goes down, indexing should be suspended.

Searching in a Fault Tolerant Distributed Configuration

Each shard will have multiple replicas. A Virtual IP (VIP) should be configured in the load balancer for each shard, consisting of all replicas. LucidWorks Search distributed search configuration, and the `shards` parameter for distributed search requests should use these VIPs.

A single VIP consisting of all the shard VIPs should be configured for all external systems to use the search service.

Integrating Monitoring Services

This functionality is
not available with
LucidWorks Search
on AWS or Azure

Monitoring your application always is an important part of running production system. Most system administrators have used various tools to ensure everything is ok from the health of server's filesystem to the the temperature of CPUs. LucidWorks Search provides additional capabilities to integrate application level statistics information into these monitoring tools.

LucidWorks Search and Solr make available several JMX MBeans which can be used with stand-alone JMX clients, or integrated with servers that support MBeans, such as Nagios or Zabbix. More information on all these options is below.

- [JMX](#)
 - [Enabling JMX for LucidWorks Search](#)
 - [JMX Clients](#)
 - [JMX MBeans](#)
- [Integrating with Monitoring Systems](#)
 - [Zabbix](#)
 - [Nagios](#)
- [Helpful Tips](#)

JMX

[JMX](#) is a standard way for managing and monitoring all varieties of software components for Java applications. JMX uses objects called MBeans (Managed Beans) to expose data and resources from your application. LucidWorks Search provides number of read-only monitoring beans that provide useful statistical/performance information. Combined with JVM (platform JMX MBeans) and OS level information, it becomes powerful tool for monitoring.

Enabling JMX for LucidWorks Search

By default JMX is enabled in LucidWorks Search for local access only. If you want to connect and monitor application remotely you need to change `lwecore.jvm.params` parameter in the `$LWS_HOME/conf/master.conf` file and add the following JVM parameters:

```
lwecore.jvm.params=... -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=3000 -Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
-Djava.rmi.server.hostname=my.server.name
```

Where 3000 is an unused TCP port number.

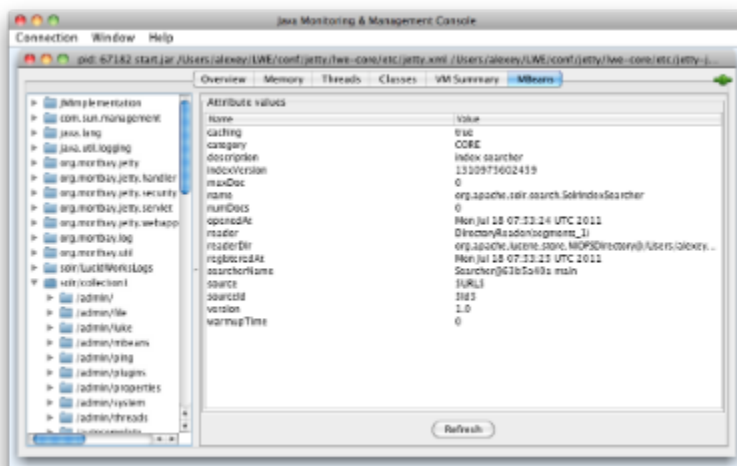
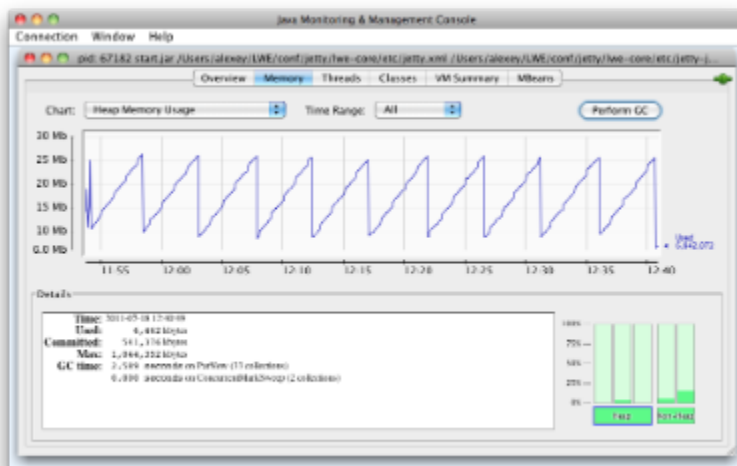
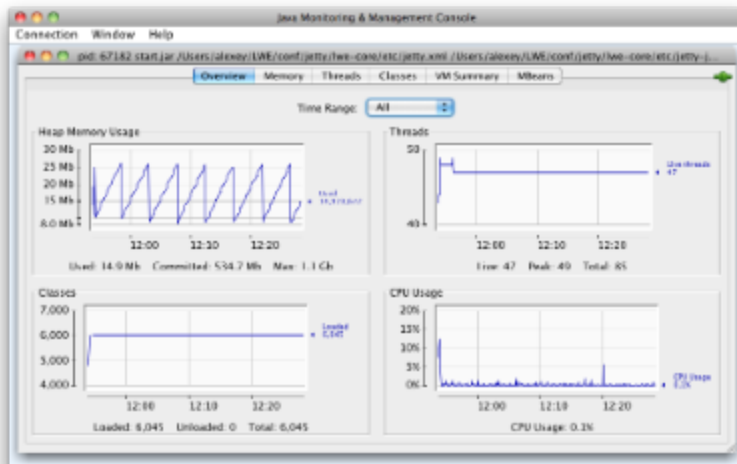
You might want to secure remote JMX access either by configuring a software or hardware firewall to allow connections to specified port only from your hosts/network or by configuring password authentication and/or SSL encryption. For more information about various security options please refer to the [JMX documentation](#).

JMX Clients

There are number of various JMX clients you can use to connect to the LucidWorks Search server and browse available information.

JConsole

[JConsole](#) is a standard (part of the JDK) graphical monitoring tool to monitor Java Virtual Machine (JVM) and Java applications which provides a nice way to display memory and CPU information as well MBeans from arbitrary applications.



JMXTerm

Jmxterm is an open source command line based interactive JMX client. It allows you to easily navigate JMX MBeans on remote servers without running a graphical interface or opening a JMX port. It can also be integrated with script languages such as Bash, Perl, Python, Ruby, etc. See the following as an example of how it can be used:

```
sh> java -jar jmxterm-1.0-alpha-4-uber.jar
Welcome to JMX terminal. Type "help" for available commands.

$>jvms
67183      ( ) - start.jar /Users/alexey/LWE/conf/jetty/rails/etc/jetty.xml
/Users/alexey/LWE/conf/jetty/rails/etc/jetty-jmx.xml
/Users/alexey/LWE/conf/jetty/rails/etc/jetty-ssl.xml
67182      (m) - start.jar /Users/alexey/LWE/conf/jetty/lwe-core/etc/jetty.xml
/Users/alexey/LWE/conf/jetty/lwe-core/etc/jetty-jmx.xml
/Users/alexey/LWE/conf/jetty/lwe-core/etc/jetty-ssl.xml
93534      ( ) - jmxterm-1.0-alpha-4-uber.jar
8554       ( ) -

$>open 67182
#Connection to 67182 is opened

$>domains
#following domains are available
JMImplementation
com.sun.management
java.lang
java.util.logging
org.mortbay.jetty
org.mortbay.jetty.handler
org.mortbay.jetty.security
org.mortbay.jetty.servlet
org.mortbay.jetty.webapp
org.mortbay.log
org.mortbay.util
solr/LucidWorksLogs
solr/collection1

$>domain solr/collection1
#domain is set to solr/collection1

$>beans
#domain = solr/collection1:
...
solr/collection1:id=collection1,type=core
solr/collection1:id=org.apache.solr.handler.StandardRequestHandler,type=standard
...
solr/collection1:id=org.apache.solr.search.FastLRUCache,type=fieldValueCache
solr/collection1:id=org.apache.solr.search.LRUCache,type=documentCache
solr/collection1:id=org.apache.solr.search.LRUCache,type=filterCache
```

```
solr/collection1:id=org.apache.solr.search.LRUCache,type=queryResultCache
solr/collection1:id=org.apache.solr.search.SolrFieldCacheMBean,type=fieldCache
...
solr/collection1:id=org.apache.solr.search.SolrIndexSearcher,type=searcher
solr/collection1:id=org.apache.solr.update.DirectUpdateHandler2,type=updateHandler

$>bean type=updateHandler,id=org.apache.solr.update.DirectUpdateHandler2
#bean is set to
solr/collection1:type=updateHandler,id=org.apache.solr.update.DirectUpdateHandler2

$>info
#mbean =
solr/collection1:type=updateHandler,id=org.apache.solr.update.DirectUpdateHandler2
#class name = org.apache.solr.core.JmxMonitoredMap$SolrDynamicMBean
# attributes
%0 - adds (java.lang.String, r)
%1 - autocommit maxTime (java.lang.String, r)
%2 - autocommits (java.lang.String, r)
%3 - category (java.lang.String, r)
%4 - commits (java.lang.String, r)
%5 - cumulative_adds (java.lang.String, r)
%6 - cumulative_deletesById (java.lang.String, r)
%7 - cumulative_deletesByQuery (java.lang.String, r)
%8 - cumulative_errors (java.lang.String, r)
%9 - deletesById (java.lang.String, r)
%10 - deletesByQuery (java.lang.String, r)
%11 - description (java.lang.String, r)
%12 - docsPending (java.lang.String, r)
%13 - errors (java.lang.String, r)
%14 - expungeDeletes (java.lang.String, r)
%15 - name (java.lang.String, r)
%16 - optimizes (java.lang.String, r)
%17 - rollbacks (java.lang.String, r)
%18 - source (java.lang.String, r)
%19 - sourceId (java.lang.String, r)
%20 - version (java.lang.String, r)
#there's no operations
#there's no notifications

$>get cumulative_adds
#mbean =
solr/collection1:type=updateHandler,id=org.apache.solr.update.DirectUpdateHandler2:
cumulative_adds = 125;
```

JMX MBeans

LucidWorks includes a number of useful JMX MBeans, some available through Solr and some developed in LucidWorks Search itself:

Solr MBeans

Domain	Objects	Available attributes	Comments
<code>solr/ collection</code>	<code>type=updateHandler, id=org.apache.solr.update. DirectUpdateHandler2</code>	<code>cumulative_adds, cumulative_deletesById, cumulative_deletesByQuery, cumulative_errors, commits, autocommits, optimizes, rollbacks, docsPending, etc</code>	This MBean provides comprehensive information about indexing activity like number of added documents, number of errors, number of commits, autocommits and optimize operations. It is really useful to plot that information into graphs in your monitoring system. The <i>cumulative_errors</i> parameter shows the number of low level IO exceptions.
<code>solr/ collection</code>	<code>type=/update, id=org.apache.solr.handler. XmlUpdateRequestHandler</code>	<code>request, errors, avgTimePerRequest, etc</code>	If using direct Solr API, there are separate beans for all types of handlers you can use to index documents into the system, such as XML, CSV, JSON request handlers. It makes sense to add this UpdateRequest Handler information to indexing graphs as well. You might also setup monitoring alert on a number of errors for particular update handler to make sure LucidWorks Search clients don't hit any errors during indexing like invalid fields names or types, no required fields in indexed documents, etc.

<code>solr/ collection</code>	<code>type=/lucid, id=org.apache.solr.handler. StandardRequestHandler</code>	<code>requests, errors, timeouts, avgTimePerRequest</code>	This MBean represents the default LucidWorks Search request handler and provides statistics about number of search requests, errors, timeouts and average response time for search requests. It's pretty useful to display this information on monitoring graphs as well as setup monitoring alerts, such as, "notify administrator if average response time is more than 0.5 second or total number of errors and timeouts is more than 1% of total requests".
-----------------------------------	--	--	---

<code>solr/ collection</code>	<code>type=searcher, id=org.apache.solr.search. SolrIndexSearcher</code>	<code>numDocs, warmupTime</code>	<p><i>numDocs</i> is the total number of documents in the index. <i>warmupTime</i> is the amount of time a new Searcher takes to warm. When LucidWorks Search commits new data into index, a new Searcher is opened and warmed. The warming operation regenerates caches from the previous Searcher instance and runs some predefined in <i>solrconfig.xml</i> queries to warm up IO filesystem cache and load Lucene FieldCache in memory. This attribute basically defines how long does it take to commit before new data will be available to users. It makes sense to monitor this parameter and setup trigger to alert the LucidWorks Search administrator if it takes more time than you expect.</p>
-----------------------------------	--	----------------------------------	---

<code>solr/ collection</code>	<code>type=filterCache, id=org.apache.solr.search. LRUCache</code>	<code>cumulative_evictions, cumulative_hitratio, cumulative_hits, cumulative_inserts, cumulative_lookups, warmupTime, etc</code>	Solr caches popular filter query (<code>fq=category:IT</code>) attributes as unordered sets of document ids. This technique significantly improves search filtering/faceting performance. <i>size</i> is the current number of cached filter queries. <i>cumulative_hitratio</i> represents if this cache is successfully utilized by giving the ratio of successful cache hits to overall number of lookups. If it's low (such as < 0.3 or 30%) over long period of time then you might want either increase cache size or disable it at all to reduce performance overhead.
<code>solr/ collection</code>	<code>type=queryResultCache, id=org.apache.solr.search. LRUCache</code>	<code>cumulative_evictions, cumulative_hitratio, cumulative_hits, cumulative_inserts, cumulative_lookups, warmupTime, etc</code>	This cache stores ordered sets of document IDs and the top N results of a query ordered by some criteria. It has the same attributes as <code>filterCache</code> .
<code>solr/ collection</code>	<code>type=documentCache, id=org.apache.solr.search. LRUCache</code>	<code>cumulative_evictions, cumulative_hitratio, cumulative_hits, cumulative_inserts, cumulative_lookups, etc</code>	The <code>documentCache</code> stores Lucene Document objects that have been fetched from disk.

LucidWorks Search MBeans

Domain	Objects	Available attributes	Comments
--------	---------	----------------------	----------

lwe	id=crawlers, name=<data_source_id>, type=datasources	total_runs, total_time, num_total, num_new, num_updated, num_unchanged, num_failed, num_deleted	This MBean displays crawlers statistics information for specific data source (like number of processed documents, number of errors, etc). If you have periodically or long running scheduled data source then you might want to monitor and alert if there's any problem with the underlying source (web site, SharePoint server, etc) or how optimized your incremental crawl is (percentage of num_unchanged to num_total), for example.
lwe	id=crawlers, name=<collection_name>, type=collections	total_runs, total_time, num_total, num_new, num_updated, num_unchanged, num_failed, num_deleted	If you have multiple data sources and don't want to monitor on per data source level, but keep an eye on aggregate numbers for the whole collection you might want to use this bean.
lwe	id=crawlers, type=total	total_runs, total_time, num_total, num_new, num_updated, num_unchanged, num_failed, num_deleted	You can use this MBean if you have multiple collections (homogeneous collections or multi-tenant architecture) to monitor on per instance level.

Integrating with Monitoring Systems

Using JConsole and JmxTerm tools is a good way to explore information hidden in JMX, but what you really need is to monitor your application automatically, record historical information, display it in a graphical form, configure parameters thresholds as triggers and send alerts in case of denial of service or performance problems. There are various standard sysadmin tools for that and integrating LucidWorks with them is no different than with any other Java application. The idea is that you can retrieve application information and send it to external monitoring system. In our documentation we provide two examples of integrating LucidWorks server with popular open source monitoring tools - [Zabbix](#) and [Nagios](#).

Zabbix

[Zabbix](#) is an enterprise-class open source distributed monitoring solution for networks and applications. It comes with pre-defined templates for almost all operating systems as well as various open source applications. It also has a great template for JVM that contains the most vital statistics of arbitrary Java application. There are different ways how you can integrate LucidWorks with Zabbix and the best approach depends on the Zabbix release version.

Pre-2.0 Releases

Zabbix does not contain built-in support for monitoring Java applications prior to v2.0, but if you are handy with scripting and command line tools then there are two possible approaches:

UserParameter: You can configure the Zabbix system agent to send custom monitored items using `UserParameter`. For retrieving JMX statistics you can use either [cmdline-jmxclient](#) or [jmxterm](#) as command line clients.

```
UserParameter=jvm.maxthreads, java -jar cmdline-jmxclient.jar localhost:3000  
java.lang:type=Threading PeakThreadCount
```

zabbix_sender utility: If you have a large number of JMX monitored items, or you need to monitor some items quite frequently, then spawning a Java Virtual Machine process to get a single object/attribute can be too expensive. In this case consider scripting JMX interactions using the [JMXTerm](#) command line tool and your favorite scripting language. The solution below is in Ruby but could be implemented using any scripting language. The main idea is that you can run a `JMXTerm` java application from your script and communicate with it using `stdin` and `stdout` streams using the [expect](#) library.

```
require "open3"
require 'expect'

....
# run jmxterm java application
stdin, stdout, wait_thr = Open3.popen2e('java -jar jmxterm-1.0-alpha-4-uber.jar')
# wait for prompt
result = stdout.expect('$>', 60)
...
# connect to specific jvm
stdin.puts("open #{process_id}")
result = stdout.expect('$>', 60)
...
stdin.puts('get -d solr/collection1 -b
type=searcher,id=org.apache.solr.search.SolrIndexSearcher numDocs')
result = stdout.expect('$>', 60)
# parse response from jmxterm command
...
# run zabbix_sender command to send single item or save multiple values into file and
send as a batch
output = `zabbix_sender -z #{@server_name} -p #{@server_port} -i file.txt`.chomp
# parse response and validate that operation is successful
...
```

2.x Releases

Zabbix 2.0 contains built-in support for monitoring Java applications (Zabbix Java proxy). For more information please see the [JMX Monitoring section of the Zabbix manual](#).

The following steps describe how to integrate LucidWorks Search with the Zabbix 2.0 release.

1. Download and install the 2.0 release according to the official [documentation](#).
2. In order to build Zabbix JMX proxy you should build Zabbix package with the `--enable-java` configuration option, such as `./configure --enable-server --with-mysql --enable-java`.

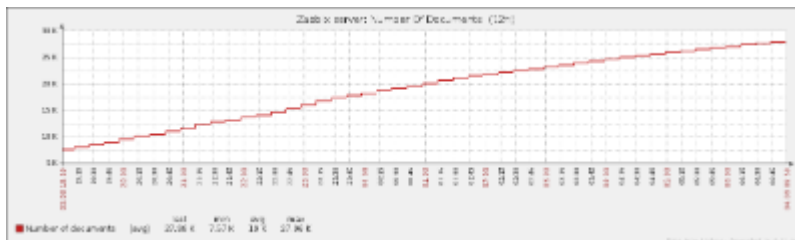
If you intend to run Zabbix on the same server where you installed LucidWorks, you may want to add the `--enable-agent` option, such as `./configure --enable-server --with-mysql --enable-java --enable-agent`.

3. After `make install`, copy the example `init.d` start script from `misc/init.d/debian/zabbix-server` into the `/etc/init.d` directory and edit it to start the JMX proxy daemon by adding `<install_dir>/sbin/zabbix_java/startup.sh` and `<install_dir>/sbin/zabbix_java/shutdown.sh` calls to the corresponding options in `init.d`.

4. Configure JMX proxy in `/etc/zabbix/zabbix_server.conf` by editing the `JavaGateway`, `JavaGatewayPort` and `StartJavaPollers` parameters. The `JavaGatewayPort` should match the `LISTEN_PORT` defined in `<install_dir>/sbin/zabbix_java/settings.sh`. It is also recommended to enable JMX proxy verbose logging by editing `<install_dir>/sbin/zabbix_java/lib/logback.xml` and changing the `file` element to point to your log file directory and setting the `level` attribute to "debug".
5. Import, using the Zabbix UI, the sample templates found in `$LWS_HOME/app/examples/zabbix` called `lwe_zabbix_templates.xml` (there are 3 in that file).
6. Install the Zabbix agent to the server where LucidWorks Search is installed and configure it to connect to the Zabbix server.
7. Add Zabbix host and assign proper template for the specific operating system (i.e., linux, freebsd, etc.).
8. Assign the imported templates (`Template_JVM`, `Template_Solr`, `Template_LWE`) to that host.
9. Enable JMX monitoring in LucidWorks and allow the Zabbix server connect to JMX interface over the network. Instructions to enable JMX monitoring are in the [Enabling JMX for LucidWorks Search](#) section of this Guide.
10. [Add the JMX interface](#) to the host where LucidWorks is installed. This is done via the Zabbix UI by creating JMX agents for each counter.
11. Start any activity in LucidWorks (such as, crawling, indexing, or serving queries) and review the graphs for the monitored host (see screenshots below).

Example graphs

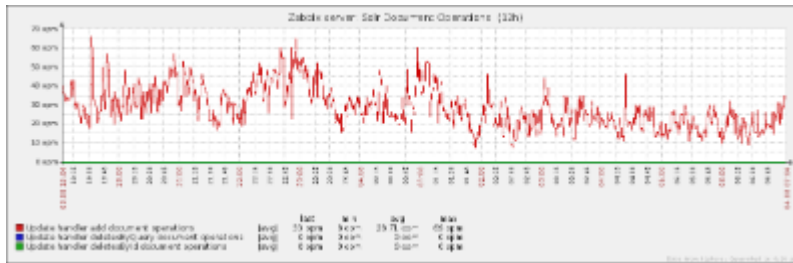
- Total number of documents in search index



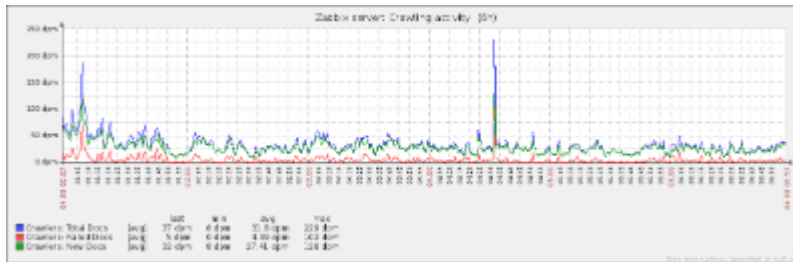
- Solr index operations (commits, optimizes, rollbacks)



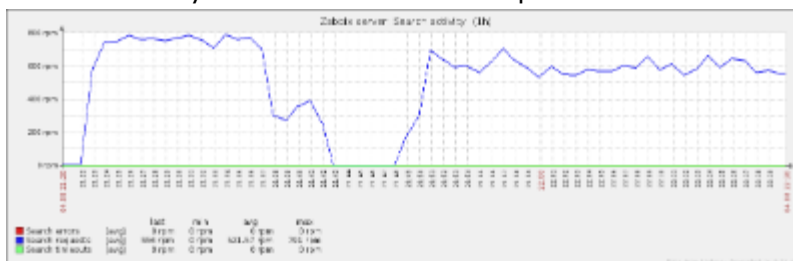
- Solr document operations (adds, deletes by id or query)



- Crawling activity - number of total documents processed, number of failures (retrieve, parsing), number of new documents



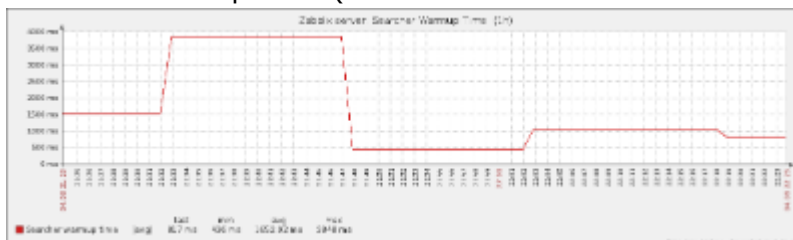
- Search activity - number of search requests



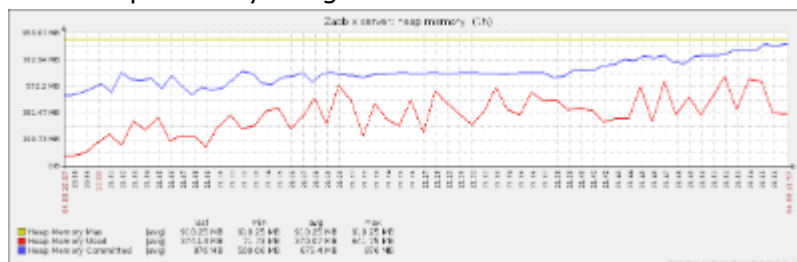
- Search Average Response Time



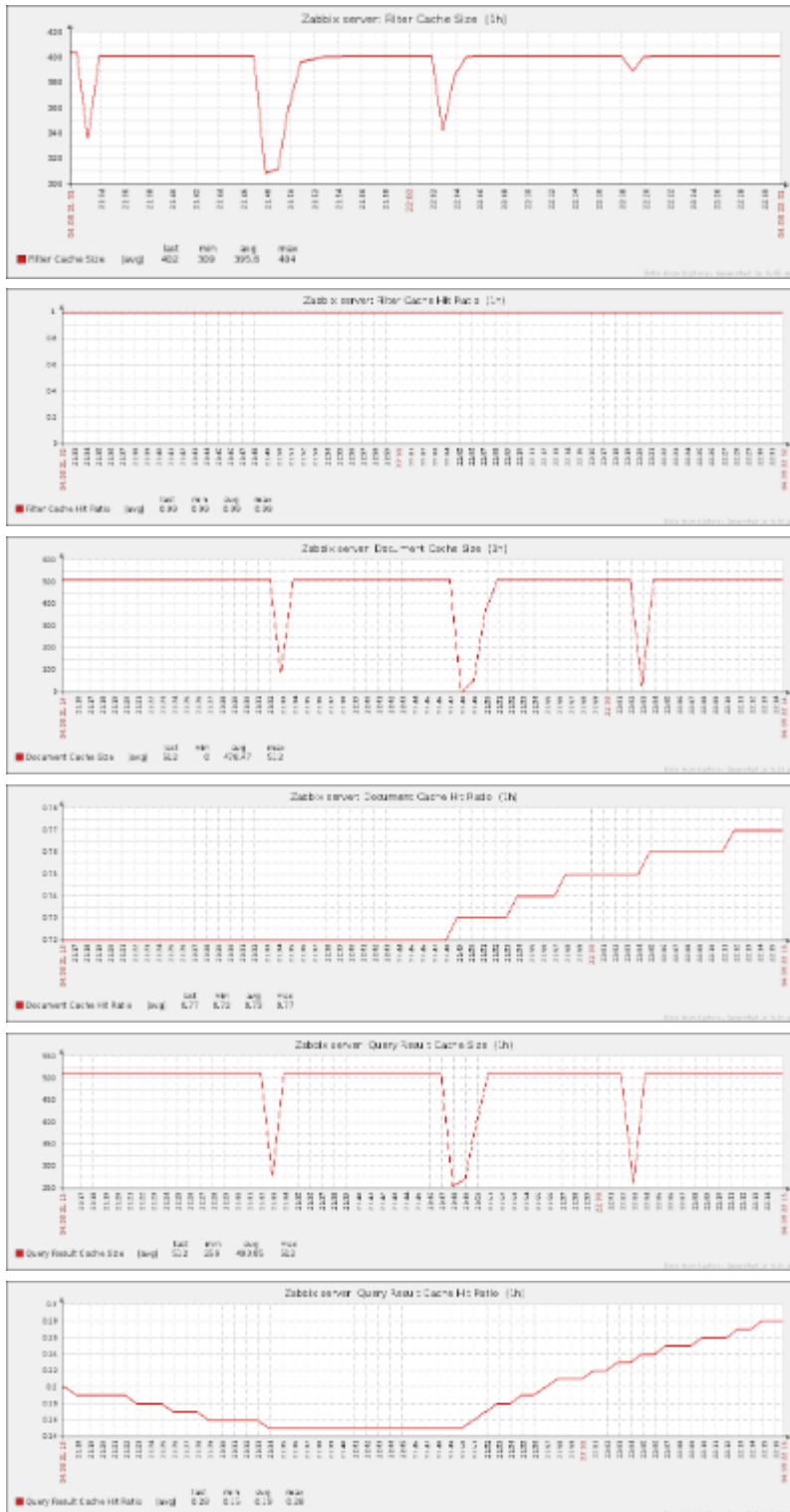
- Searcher Warmup Time (how fast committed docs become visible/searchable)



- Java Heap Memory Usage



- Caches stats



Nagios

[Nagios](#) is a popular open source computer system and network monitoring software application. It watches hosts and services, alerting users when things go wrong and again when they get better. There are different Nagios plugins that allow you to monitor Java applications using JMX interface. We recommend you to use [Syabru Nagios JMX Plugin](#) as the most mature plugin that supports different data types (integers, floats, string regular expressions) and advanced Nagios threshold syntax. In order to install Syabru Nagios JMX Plugin you should copy `check_jmx` and `check_jmx.jar` from the downloaded package to Nagios `plugins` directory and add [check_jmx command definition](#) to either global `commands.cfg` configuration file or put the `jmx.cfg` file into `nagios_plugins` configuration directory. The next step is to define Nagios services, as in this example:

```
# LWE searcher warmup time is no more than 1) 1 second - warning state 2) 2 seconds -
critical state
define service {
    hostgroup_name          all
    service_description      LWE_SEARCHER_WARMUP_TIME
    check_command            check_jmx!3000!-O
    "solr/collection1:type=searcher,id=org.apache.solr.search.SolrIndexSearcher" -A
    warmupTime -w 1000 -c 2000 -u ms
    use                      generic-service
    notification_interval    0
}

# LWE search average response time is no more than 1) 100ms - warning state 2) 200ms -
critical state
define service {
    hostgroup_name          all
    service_description      LWE_SEARCHER_AVG_RSP_TIME
    check_command            check_jmx!3000!-O
    "solr/collection1:type=/lucid,id=org.apache.solr.handler.StandardRequestHandler" -A
    avgTimePerRequest -w 100 -c 200 -u ms
    use                      generic-service
    notification_interval    0
}
```

After you setup your services and reload the Nagios configuration you can monitor application state using either the Nagios web UI or receive email notifications.

- Nagios UI screenshot (thresholds on the screenshots are lowered to trigger critical state as an example)

Host ↑↓	Service ↑↓	Status ↑↓	Last Check ↑↓	Duration ↑↓	Attempt ↑↓	Status Information
localhost	LWE_SEARCHER_AVG_RSP_TIME	CRITICAL	2011-08-22 08:02:21	0d 0h 23m 52s	4/4	JMX CRITICAL - avgTimePerRequest = 3.3344653ms
	LWE_SEARCHER_WARMUP_TIME	CRITICAL	2011-08-22 08:01:56	0d 0h 17m 53s	4/4	JMX CRITICAL - warmupTime = 1294ms

- Nagios email alert

**** PROBLEM Service Alert: localhost/LWE_SEARCHER_WARMUP_TIME is CRITICAL ****

Inbox | X

★ nagios@ip-10-110-235-82.ec2.internal to me [show details](#) 11:52 AM (44 minutes ago)

***** Nagios *****

Notification Type: PROBLEM

Service: LWE_SEARCHER_WARMUP_TIME

Host: localhost

Address: 127.0.0.1

State: CRITICAL

Date/Time: Mon Aug 22 07:52:01 UTC 2011

Additional Info:

JMX CRITICAL - warmupTime = 1114ms

Helpful Tips

- **OS file system cache:** One of the frequent problems with LucidWorks Search and Lucene/Solr applications is that if you do not have enough free memory and a significant index size you might notice performance problems because there's not enough free memory for the file system cache. IO cache is a crucial resource for search applications, so it definitely makes sense to monitor this parameter and display it in graphs with other memory information like free memory, jvm heap memory, swap, etc. This parameter is part of the OS level monitoring in Zabbix (name is `vm.memory.size[cached]`).
- **File descriptors:** Another problem is that sometimes your application can hit OS or per process file descriptor limits. It is also recommended to monitor these parameters and set trigger thresholds for these parameters.
- **CPU usage:** Default Zabbix templates have triggers for CPU load average numbers. You might want to tune thresholds for your server based on number of CPUs and expected load.
- **Heap memory usage and garbage collector statistics:** Zabbix Java template contains multiple items and triggers for memory and garbage collector invocation counts. You should also tune these parameters to match your scenario.
- **Solr index size and free disk space:** These should be set properly to avoid "Out Of Disk Space" errors.

Glossary of Terms

Where possible, terms are linked to relevant parts of the documentation 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

Alerts

An alert allows a user to save searches. There are two types: *active*, which will send notifications when new results are found, and *passive*, which do not send notifications.

Auto-Complete

A way to provide users suggestions for possible matching queries before they have finished typing. In LucidWorks Search, this relies on an index of terms to be created on a regular basis by scheduling it as an activity.

B

Boolean Operators

These control the inclusion or exclusion of keywords in a query by using operators such as AND, OR, and NOT.

C

Click Scoring Relevance Framework

A method of changing the relevance ranking of a document based on the number of times other users have clicked on the same document.

Collection

One or more documents grouped together for the purposes of searching. See also [Document](#).

Component

A part of LucidWorks Search that has been designed to stand alone or can be run independently from other components. LucidWorks Search has three main components: *LWE-Core*, which runs Solr, indexing, and other critical application functions, *LWE-Connectors*, which handles all crawling activities, and *LWE-UI*, which runs the Administrative UI, the front-end search interface, and the alerting functionality.

Connector

A connector is a program or piece of code that allows a connection to be made to a data source and content to be extracted from it.

Crawler

Also known as a "spider", this is a program that is able to retrieve documents internal or external servers.

D ---

Data Source

Defines the metadata required to connect to a location containing content to be indexed. It could be a file system path, a Web URL, a JDBC connection, or some other set of values.

Distributed Index

A distributed index is one where the search index for a [collection](#) is spread across more than one [shard](#).

Distributed Search

Distributed search is one where queries are processed across more than one [shard](#).

Document

One or more Fields. See also [Field](#).

F ---

Field

The content to be indexed/searched along with metadata defining how the content should be processed by LucidWorks Search.

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 http://lucene.apache.org/core/old_versioned_docs/versions/3_5_0/scoring.html 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.

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?"

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.

Replication

A method of copying a master index from one server to one or more "slave" or "child" servers. In LucidWorks Search, the master continues to manage updates to the index, while queries are handled by the slaves. This approach enables LucidWorks Search to properly manage query load and ensure responsiveness.

REST API

An alternative way of controlling LucidWorks Search without accessing the user interface.

S

Shard

A method of partitioning a database or search engine to maximize performance and efficiency.

SolrCloud

[Ongoing work](#) within the Solr community to improve Solr's ability to operate in a cloud environment.

Solr Schema (schema.xml)

The Apache Solr index schema. The schema defines the fields to be indexed and the type for the field (text, integers, etc.) The schema is stored in schema.xml and is located in the Solr home conf directory.

Solr Config (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. In LucidWorks Search, when spell-checking is enabled, a parallel "spell" index is created as documents are indexed.

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")

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 http://lucene.apache.org/java/2_3_2/scoring.html for more info on TF-IDF based scoring and Lucene scoring in particular.

See also [Inverse Document Frequency \(IDF\)](#).

W

Wildcard

A wildcard allows a substitution of one or more letters of a word to account for possible variations in spelling or tenses. In LucidWorks Search, there are two ways to use them. One is to use an asterisk (*) at the end of a term to find all documents that contain words that start with that pattern. For example, `paint*` would find `paint`, `painter` and `painting`. A second way is to use a question mark (?) in the middle of a term to substitute for one character in that term. Such as, `c?t` would find `cat`, `cot` and `cut`. It's also possible to use wildcards at the start of a term in the same way - either to replace a single letter (using the ? symbol) or to find documents that contain words that end with a pattern using a *. For example, `*sphere` would find `ecosphere` and `stratosphere`.

About LucidWorks

LucidWorks (formerly known as Lucid Imagination) is the trusted name in Search, Discovery and Analytics, delivering the only enterprise-grade embedded search development solution built on the power of the Apache Lucene/Solr open source search project. Founded in 2008, the company initially provided support, consulting services, documentation and training for the Apache Lucene/Solr open source search project.

Within a few years, the LucidWorks team realized the need to add value to the open source search platform by developing an extensive layer of services which made Lucene/Solr secure and easier to use and manage. The company shipped the first version of its flagship product, LucidWorks Search, in 2011, followed by LucidWorks Big Data in May 2012. LucidWorks continues to offer support, documentation, consulting services and training products for Lucene/Solr.

LucidWorks remains committed to giving back to the Apache Lucene/Solr community. Out of the 37 Core Committers to the Apache Lucene/Solr project, 9 individuals work for LucidWorks, making the company the largest supporter of open source search in the industry. Further, LucidWorks hosts the Lucene Revolution, a conference dedicated to sharing ideas and promoting the Apache Lucene/Solr open source search project.

For more information on product and support options for LucidWorks Search, please write to: sales@lucidworks.com or visit our [website](#). Support inquiries can be submitted to our [Support group](#).



[LucidWorks](#)

3800 Bridge Parkway, Suite 101
Redwood City, CA 94065

Tel: 650.353.4057
Fax: 650.525.1365