

# LucidWorks Search Lucid Query Parser Guide

## 2.7 Documentation

# Table of Contents

---

How to Use this Documentation	5
Audience and Scope	5
Conventions	5
Customers of LucidWorks Search on AWS or Azure	7
Getting Support & Training	8
Lucid Query Parser	9
Features	9
Building Search Queries	10
Building Advanced Queries	52
Query Parser Customization	79
About LucidWorks	84

# 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

## 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

## Lucid Query Parser

- [How the default query parser handles user requests](#)
- [Customization options](#)

## **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

## **Custom Connector Guide**

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

# 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
-----------	--------------------

Note Type	Look & Description
Information	 Notes with a blue background are used for information that is important for you to know.
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	 <b>Information for LucidWorks Search in the Cloud Users</b> 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.

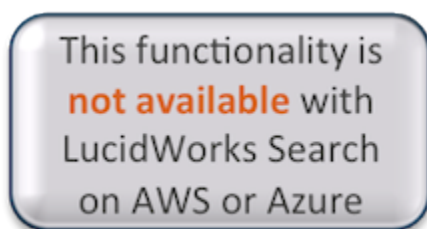
Customers hosted on AWS or Azure should see the section for [#Customers of LucidWorks Search on AWS or Azure](#) below.

## 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-XXXXXXXX.lucidworks.io`" where XXXXXXXX is 8 characters (letters or numbers). So, if your instance URL is "`https://s-9sdff10b.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](#).



# Lucid Query Parser

---

The Lucid query parser is designed as a replacement for the existing Apache Lucene and Solr query parsers to produce good results with queries in any format, extracting the maximum possible information from the user entry to produce the best matches. The user (or system administrator) does not need to indicate what type of query it is: query interpretation is "modeless". The system will never produce an error message in response to a query and will always make the best interpretation possible. The system will match documents with alternative word forms (for example, singulars/plurals; this is on by default), expand synonyms, spell-check queries and handle queries in various languages (English by default) as configured by the administrator.

## Features

---

The basic features of the Lucid query parser include all those of the traditional Apache Lucene and Apache Solr DisMax query parsers and is designed to work well for users accustomed to the search syntax used by popular Web search engines.

The Lucid query parser also includes a number of features not found in some or all of these other query parsers, including:

- [Relational operators](#) (comparative operators) - '<', '<=', '==', '!=', '>=', '>'
- More flexible [range searches](#)
- Implicit AND operator when no explicit operators are present
- The ALL pseudo-field to search across all fields
- Automatic bigram and trigram relevancy boosting
- Support for [natural language queries](#)
- Support for a wide variety of [date formats](#)
- Support for [advanced proximity](#) operators - NEAR, BEFORE, AFTER
- Support for multi-word synonyms
- Enhanced support for [hyphenated terms](#)
- Case-insensitive [field names](#)
- Sticky field names
- Good results even in the presence of query syntax errors with automatic error recovery
- Intelligent out-of-the-box settings for best results
- Wide range of configuration options for special needs

Simply put, the Lucid query parser is capable of handling existing Lucene and Solr queries and many Web search engine queries.

## Building Search Queries

---

This section reviews some of the most commonly used types of search queries and gives examples of how LucidWorks processes them. More information can be found in the section on [Building Advanced Queries](#).

## Basic Usage

The Lucid query parser uses a number of built-in but configurable heuristics to automatically detect and process queries in a variety of formats.

*Basic keyword* queries are made up of words and quoted phrases, such as those used in Web search engines. As in those search engines, the default interpretation rule for such queries is that all significant words in the query must be present for a document to match. Examples:

- quantum physics
- address of "White House"

*Simple Boolean* queries are made up of words or phrases preceded by a '+' to indicate that a word or phrase must be present for a document to match, or by a '-' to mean they cannot be present. If a '+' or '-' or relational operator is present, other words or phrases in the query are interpreted as 'nice to have' but are not required or excluded for documents to match. For example,

- +pet cat dog fish rabbit -snakes

*Full Boolean* queries use any legal combination of the Boolean operators AND, OR, NOT and an unlimited number of matching parentheses. The AND and OR operators may be any case, but the NOT operator must always be upper case. Configuration settings can be used to alter these defaults. For example,

- (dog AND puppy) OR (cat AND kitten)
- lincoln and washington NOT (bridge or tunnel)

*Extended Boolean* queries combine simple Boolean queries (a term list or list of terms and quoted phrases, optionally using the '+' and '-' operators) with the Boolean AND, OR, and NOT operators. For example,

- Abraham +Lincoln -tunnel or George +Washington -bridge and (early history or influences)

*Natural language* queries, are made up of either sentence fragments or natural language questions that begin with one of the question words 'Who', 'What', 'When', 'Where', 'Why', or 'How' and end with a question mark. For example,

- the return of the pink panther
- what is aspirin?
- How does a rocket work?

Queries using Lucene syntax (see below) include field delimiters, Proximity Operators, wildcards, and so on. This syntax may be used with arbitrarily complex Boolean expressions. For example,

- title:psycholog\*
- [1968 TO 1972]

- "software development"~10

*More like this* queries are those in which the users types or pastes in a long section of text as a model to match. The Lucid query parser uses a statistical approach for matching such queries.

For queries in any other format, the Lucid query parser will do a best effort interpretation.

Although these features work together automatically out of the box, a wide range of configuration options are available and described in the following sections.

Users should use quotation marks around any series of words they intend to be interpreted as a literal phrase in order to find documents that must have that exact phrase. However, in most cases users are better served by omitting the quotation marks, since the query parser ranks higher those documents that have consecutive words near each other while also bringing back potentially relevant documents that have those words but not necessarily as a literal phrase.

By default, the Lucid query parser also pays attention to every word in the query and tries to use each word appropriately. Common words such as *a*, *the*, *to*, etc. (often called "stop words"), are not ignored but are treated specially. For example, a query consisting of only stop words will, as the default, require all of them to be present. On the other hand, a query such as *what is aspirin*, will only require the word *aspirin*, but make use of all the words in the query to rank the best documents highest when documents are returned in order of relevancy.

## Error Handling

The Lucid query parser is designed to be able to handle and give good results for even the most malformed queries. No exceptions will be thrown in any event. Common mistakes include mismatched parentheses, brackets, or braces, unterminated strings, missing operand for a Boolean operator, or any other operator, unknown field names, and extraneous punctuation.

In the worst case, the offending character, operator, or term will be discarded.

## Understanding Terms

The basic unit of a query is a *term*. In its simplest form, a term is simply a *word* or a *number*. A term may also include embedded punctuation such as hyphens or slashes, and may in fact be more than one word separated by such embedded punctuation. A term may also include *wildcard* ('?' and '\*').

This basic form of term is referred to as a *single term*. For example, the following are all single terms:

- cat
- CD-ROM
- 123
- 1,000
- -456
- +7.89
- \$1,000.00
- cat\*
- ?at
- at?c\*he

Numbers optionally may have a leading + or - sign that is considered part of the number. If you wish to place the 'must' or 'must not' operators in front of an unsigned number, add an extra '+' between the operator and the unsigned number.

There is a second form of term called a *phrase*, which is a sequence of terms enclosed in double quotation marks. The intention is that the terms (words) are expected to occur in that order and without intervening terms. An alternative purpose is simply to indicate that a Lucid keyword operator should be interpreted as a natural language word rather than as an operator. For example:

- "In the beginning" "George Washington" "AND" "myocardial infarction"

A third form of term is the *range query*, which is a pair of terms which will match all terms that lexically fall between those two terms. For example, [cat TO dog] matches all terms between cat and dog in lexical order (i.e., alphabetically, in this case).

A fourth form of term is a parenthesized sub-query which may be a complex Boolean query. For example:

- cat (bat OR fish AND giraffe) zebra

A sequence of terms (of any form – single, phrase, range, or sub-query) is referred to as a *term list*. A term list might be used to represent one or more compound terms, or simply a list of keywords and phrases, or a combination of the two. A term list is the most common form of query. In this basic form, a term list has no operators.

Note that for non-text fields, a term may not actually be a word or number, but may have a special syntax specific to that field such as a part number or telephone number.

In order to offer advanced search functions, each term can optionally be preceded or followed by various *term modifiers*. An example of a modifier before a term is a *field name*. An example of a modifier after a term is a *boost factor*.

A subset of operators (*term operators*, + and -) may also be included within a term list.

## Case Insensitivity

As a general rule, query text is *case insensitive*, meaning that you may use any combination of upper and lower case letters regardless of the case used in documents in the search collection. This also includes field names and keyword options. Mixed case may be used for technical correctness (e.g., proper names) or for readability or emphasis, but will in no way affect the query results.

One exception is in *string* or *keyword* fields, as distinct from *full-text* fields, where the administrator may have chosen to maintain precise case for precision or some other reason.

Another exception is that the administrator may decide to remove the term analysis filter which is responsible for making sure that terms are converted to lower case when they are indexed as well as query time.

But in general you should feel free to enter queries as feels most natural and feel free to enter the entire query in lower case if that is what feels most natural to you.

The following queries will be interpreted identically:

- `president george washington "cherry tree" datemodified:1994`
- `President George Washington "Cherry Tree" dateModified:1994`
- `president GEORGE Washington "CHERRY TREE" DateModified:1994`
- `PRESIDENT george WASHINGTON "CHERRY tree" DATEMODIFIED:1994`

## Simple Boolean Queries

A simple Boolean query consists of a list of terms, single keywords or quoted phrases, some of which may be preceded with the '+' and '-' Boolean operators. Terms preceded by the '+' operator are *required*, meaning that only documents containing those terms will be selected by the query. Terms preceded by the '-' operator are *prohibited*, meaning that only documents that do not contain any of the prohibited terms will be selected by the query. All other terms are considered *optional*, meaning that none of those terms are required for a document to be selected by the query, but documents containing those terms will be ranked higher based on how many of the optional terms they contain. For example,

- `president george +washington -bridge`

Which selects documents which contain "Washington" but not "bridge". Selected documents do not have to contain "president" or "george", but they will be ranked higher if they do.



## Natural Language Queries

A *natural language* query is one in which the user does not use any special syntax, but instead enters their search in the form of a statement or question, or as they would normally speak to another person. In general, the best first stab at any query is to write the query as a natural language question, such as:

- `what is aspirin?`

Or, include all of the connective words to fully express the topic so that they can participate in relevancy boosting. For example, rather than asking a user to conform their query to the query parser with:

- `title: return "pink panther"`

the user can simply enter the title as it would normally appear:

- `the return of the pink panther`

The Lucid query parser will automatically reformulate the query so that documents containing the exact wording will rank high, but documents containing subsets of the query will rank reasonably high as well.

A special feature supporting natural language queries is that a trailing question mark ('?', which normally would be treated as a wildcard character) will automatically be stripped from the query if there are at least three terms in the query and the initial term is a question word, "who", "what", "when", "where", "why", or "how". If for some reason you do wish to enter a query that ends with a wildcard question mark but starts with a question word, simply follow it with a space and a period or other punctuation character. For example,

- `what is aspirin? .` (Treated as a wildcard)

The exact minimum word count is configurable with the `minStripQMark` configuration setting.

## Phrase Query

In natural language, a phrase is simply a sequence of words or terms. Query languages usually require that the phrase be enclosed within double quotation marks. Raw natural language text is frequently as good as explicitly quoted phrases. However, in some cases an explicit phrase may be best for a particular query.

A quoted phrase is simply any query text (or an n-gram) enclosed within double quotation marks. For example:

- "John Doe"
- "John Q. Doe"
- "Java software development"
- "The quick brown fox jumped over the lazy dog's back."

The text may consist of words as well as punctuation. Although operators and other special characters may also be present, they will not have their usual meaning and will be considered as words or punctuation. In particular, wildcards, parentheses, "+" and "=", and boost factors have no special meaning inside of a quoted phrase. For example, the following queries are equivalent:

- "--The +cat (in the hat?), ran^2.0 \* -away."
- "The cat in the hat ran 2.0 away"

An empty phrase query (that is, "") will be ignored.

A phrase query containing a single term will be treated as a single term query, but any wildcards or operators within the term will be ignored and stop words will be processed as non-stop words.

In addition to simple Phrase Queries, the Lucid Query Parser also supports [Advanced Proximity Queries](#).

## More Like This

A *more like this* query is a passage of natural language text that has more than a threshold number of terms (the default is 12, but can be configured by the administrator with the `likeMin` configuration setting in `solrconfig.xml`). All terms will be implicitly ORed. This may result in a large number of results, but automatic bigram and trigram relevancy boosting will tend to rank results that more closely match the query text. Two applications of this feature are to detect plagiarism and derivative works or subtle variations. Exact matches will rank highest, but close matches will also rank high. For example:

When in the Course of human events, it becomes necessary for one people to dissolve the political bands which have connected them with another, and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

All of that text would typically be pasted into the query box as one line of text.

Be sure not to enclose the passage in double quotation marks, otherwise the query will be treated as a precise phrase query. There is no harm in using the quoted phrase, other than that an exact match must be made in that case.

This feature may be suppressed by simply enclosing the entire query within parentheses:

(When in the Course of human events, it becomes necessary for one people to dissolve ... )

## Boolean Operators

Although the majority of queries can be constructed without resorting to explicit Boolean operators, enterprise users sometimes do need the extra power to form complex queries to narrow searches. The basic Boolean operators are AND, OR, and unary and binary NOT. Evaluation is left to right, but parenthesis can be used to control the evaluation order. For example,

- Cat OR dog AND pet NOT zebra
- cat OR NOT dog
- cat AND NOT dog
- cat NOT dog
- (cat OR dog) AND (table OR chair)
- hit AND run
- hot OR not
- NOT cat AND dog
- cat AND (NOT dog)

Binary 'NOT' is equivalent to 'AND NOT' unless preceded by 'OR'.

A Boolean operator is used to combine the results of two term lists or the results of another boolean operator. For example,

- Abraham Lincoln OR George Washington
- second-hand furniture AND (table OR chair)

Parentheses are not required around term lists (any sequence of terms without Boolean operators, also known as an *extended Boolean* query). For example, the following are equivalent to the preceding examples:

- (Abraham Lincoln) OR (George Washington)
- (second-hand furniture) AND (table OR chair)

Although Boolean operators are normally written in all upper case, lower case 'and' and 'or' are also permitted by default. Sometimes that may cause a query to be misinterpreted, but usually with little or no harm. Lower case 'not' is not permitted since it would cause very wrong results if it happened to occur as a query term. For example:

- Abraham Lincoln and George Washington
- second-hand furniture and (table or chair)
- hit and run
- hot or not

The `upOp` configuration setting can be enabled to require all Boolean operators to be upper case-only. The `notOp` configuration setting can be disabled to allow lower case 'not' as a Boolean operator. See [Query Parser Customization](#) for more details on how to change these settings.

In addition to the Boolean keywords, non-keyword operator equivalents are available as substitutions for the keyword Boolean operators. A double ampersand ('&&') means 'AND', a double vertical bar ('||') means 'OR' and a single exclamation point ('!') means 'NOT'. So, the above examples can also be written as:

- Cat || dog && pet ! zebra
- cat || ! dog
- cat && ! dog
- cat ! dog
- (cat || dog) && (table || chair)
- Abraham Lincoln || George Washington
- second-hand furniture && (table || chair)
- (Abraham Lincoln) || (George Washington)
- (second-hand furniture) && (table OR chair)
- hit && run
- hot || not
- ! cat && dog
- cat && (! dog)

Sticky field names or keyword options remain in effect only until either a new sticky field name (or keyword option) or a right parenthesis at the current parenthesis nesting level. For example,

User Entry	Equivalent
title:cat nap OR dog bark	title:(cat nap) OR title:(dog bark)
title:cat nap OR body: dog bark	title:(cat nap) OR body:(dog bark)
(body:cat AND ((title:dog) OR fish)) AND bat	(body:cat AND ((title:dog) OR body:fish)) AND default:bat

## Left to Right Boolean Evaluation Order

Unlike Lucene and Solr, the Lucid query parser assures that Boolean queries without parentheses will be evaluated from left to right. For example, the following are equivalent:

- cat OR dog OR fox AND pet
- (cat OR dog OR fox) AND pet
- ((cat OR dog) OR fox) AND pet

## Implicit AND versus Implicit OR

If none of the terms of a term list have explicit operators (+ or -, the individual terms will be implicitly ANDed into the query. Lucene and Solr default to implicit ORing of terms, but implicit ANDing tends to produce better results in most applications for most users. For example:

User Input	Query Interpreted as
heart attack	heart AND attack
George Washington	George AND Washington
Lincoln's Gettysburg Address	Lincoln's AND Gettysburg AND Address

But if one or more of the terms in a term list has an explicit term operator (+ or - or relational operator) the rest of the terms will be treated as "nice to have." For example,

- `cat +dog -fox`

Selects documents which must contain "dog" and must not contain "fox". Documents will rank higher if "cat" is present, but it is not required.

- `cat dog turtle -zebra`

Selects all documents that do not contain "zebra". Documents which contain any subset of "cat", "dog", and "turtle" will be ranked higher.

The `defOp` configuration setting in `solrconfig.xml` can be used to disable the implicit AND feature, but it is enabled by default.

## Strict vs. Loose Extended Boolean Queries

A *strict query* or Boolean query is one in which explicit Boolean operators are used between all terms. A loose query, also known as an extended Boolean query, uses a combination of explicit Boolean operators and term lists in which the operators are implicit. Put simply, extended Boolean queries allow free-form term lists as operands for the Boolean operators, while strict Boolean queries permit only a single term or quoted phrase (or parenthesized sub-query.) Loose, extended Boolean queries provide every bit of the power of a strict Boolean query, but are more convenient to write and can be easier to read. In fact, queries written in more of a natural language format with fewer explicit Boolean operators facilitate relevancy boosting of adjacent terms.

Examples of strict Boolean queries	The equivalent loose, extended Boolean queries
<code>cat AND dog</code>	<code>cat dog</code>
<code>(cat OR dog) AND (food OR health)</code>	<code>(cat OR dog) AND (food OR health)</code>
<code>cat OR dog NOT pets</code>	<code>cat dog -pets</code>
<code>(George AND Washington) OR (Abraham AND Lincoln)</code>	<code>George Washington OR Abraham Lincoln</code>
	<code>"George Washington" OR "Abraham Lincoln"</code>

<b>Examples of strict Boolean queries</b>	<b>The equivalent loose, extended Boolean queries</b>
"George Washington" OR ("Abraham" AND "Lincoln")	

## Hyphenated Terms

Hyphenated terms, such as `plug-in` or `CD-ROM`, are indexed without their hyphens, both as a sequence of sub-words and as a single, combined term which is the concatenation of the sub-words. That combined term is stored at the position of the final sub-word. Users authoring documents are not always consistent on whether they use the hyphens or not, but the goal of the Lucid query parser is to be able to match either given a query of either. To do this as well as possible, the Lucid query parser will expand any hyphenated term into a Boolean OR of the sub-words as a phrase and the combined term.

### Simple Hyphenated Terms

A query of `plug-in` will automatically be interpreted as `("plug in" OR plugin)`. If we have these mini-documents:

- Doc #1: This is a plugin.
- Doc #2: This is the plug-in.
- Doc #3: Where is my plug in?

The query will match all three documents.

A query of `plugin` will only match the first two documents, but that is a limitation of this heuristic feature. The query results are better than without this feature even if they are still not ideal.

### Hyphenated Terms within Quoted Phrases

Quoted phrases may contain any number of hyphenated terms, in which case the Lucene "span query" feature is used for the entire phrase as well as the individual hyphenated terms which are expanded as above.

A query of:

- `"buy a cd-rom with plug-in software"`

would match any of the following mini-documents:

- Doc #1: I want to buy a cdrom with plugin software
- Doc #2: I want to buy a cdrom with plug-in software
- Doc #3: I want to buy a cd-rom with plugin software
- Doc #4: I want to buy a cd-rom with plug-in software

In terms of the new proximity operators, this query is equivalent to:

- `buy a before:0 cd-rom before:0 with before:0 plug-in software`

which is equivalent to:



- `buy a before:0 ("cd rom" or cdrom) before:0 with before:0 ("plug in" or plugin) before:0 software`

## Multiple Hyphens in Terms

Some hyphenated terms have more than two sub-words. For example:

- `on-the-run` and `never-to-be-forgotten`

will be interpreted as:

- `("on the run" OR ontherun)` and `("never to be forgotten" OR nevertobeforgotten)`

Multiple hyphens occur in various special formats, such as phone numbers. For example:

- `646-414-1593 1-800-555-1212`

which will be interpreted as:

- `("646 414 1593" OR 6464141593) AND ("1 800 555 1212" OR 18005551212)`

Social Security numbers and ISBNs also have multiple hyphens. For example,

- `101-23-1234` and `978-3-16-148410-0`

will be interpreted as:

- `("101 23 1234" OR 101231234)` and `("978 3 16 148410 0" OR 9783161484100)`

Part numbers and various ID formats also tend to contain more than one hyphen. These would be treated similarly to the examples above.

## Punctuation and Special Characters

In general, any punctuation or special character that is not a query operator is treated as if it were white space. This includes commas, periods, semi-colons, slashes, etc. Punctuation or special characters before and after either a single term or the terms within a phrase will be ignored. Punctuation is sometimes referred to as a *delimiter*. For example,

User Input	Query Interpreted as
/this/	this (note: still treated as a stop word)
cat, dog; fox.	cat dog fox
Yahoo!	Yahoo
C++	C
B-	B

However, punctuation embedded within a single term or the terms of a phrase will be treated as if it were a hyphen and the term is treated as if it were a phrase with white space in place of the punctuation. For example,

User Input	Query Interpreted as
x,y,z	"x y z"
Jan/Feb/Mar	"Jan Feb Mar"
;Jan/Feb/Mar.	"Jan Feb Mar"
Jan&Feb	"Jan Feb"
"Reports for Jan&Feb"	"Reports for Jan Feb"
C++/C#/Java	"C C Java"
AT&T	"AT T"
U.S.	"U S"

Dollar signs, commas, and decimal points are treated similarly. For example,

User Input	Query Interpreted as
1,000	"1 000"
\$1,275.34	"1 275 34"

Web URLs are not treated specially, other than to allow the colon rather than treating it as a field name, so the URL special characters are removed using the same punctuation removal rules as any other term. For example,

User Input	Query Interpreted as
<a href="http://www.cnn.com/">http://www.cnn.com/</a>	"http www cnn com"
<a href="http://people.apache.org/list_A.html">http://people.apache.org/list_A.html</a>	"http people apache org list A html"

Similarly, email addresses have no special treatment, other than to treat all special characters, including the "@" and dots as delimiters within a phrase. For example,

User Input	Query Interpreted as
joe@cnn.com	"joe cnn com"
joseph.smith@whitehouse.gov	"joseph smith whitehouse gov"

## Alphanumeric Terms

Alphanumeric single terms and terms within phrases are split into separate terms as a phrase. For example:

User Input	Query Interpreted as
A20	"A 20"
B4X3	"B 4 X 3"
Alpha7	"Alpha 7"
"Nikon Coolpix P90"	"Nikon Coolpix P 90"
24x Zoom Z980	"24 x" Zoom "Z 980"

## Wildcard Queries

Any term in a query, except for quoted phrases, may contain one or more wildcard characters. Wildcard characters indicate that the term is actually a pattern that may match any number of terms. There are two forms of wildcard character: asterisk ("\*") which will match zero or more arbitrary characters and question mark ("?") which will match exactly one arbitrary character.

### Wildcards Within or At End of Terms

A term consisting only of one or more asterisks will match all terms of the field in which it is used. For example, `title:*`.

The most common use of asterisk is as the last character of a query term to match all terms that begin with the rest of the query term as a prefix. For example, `paint*`.

One traditional use of asterisk is to force plurals to match. This use is usually unnecessary because LucidWorks uses a stemming filter to automatically match both singular and plural forms. However, this technique may still be useful if the administrator chooses to disable the stemming filter or for fields that may not have a stemming filter. For example, `Sneaker*` will match both "sneaker" and "sneakers".

A question mark can be used where there might be variations for a single character. For example:

User input	Matches
?at	"cat", "Bat", "fat", "kat", and so on
c?t	"cat", "cot", "cut"
ca?	"cab", "can", "cat", and so on

Any combination of asterisks and question mark wildcards can be used in a single term, but care is needed to avoid unexpected results.

Note that wildcards are not supported within quoted phrases. They will be treated as if they were white space. Wildcards can be used for non-text fields.

If you need to use a non-wildcard asterisk or question mark in a non-text field, be sure to escape each of them with a backslash. For example,

```
myField:ABC\*DEF\?GHI
```

will match the literal term "ABC\*DEF?GHI".

If you need to use a trailing question mark wildcard at the end of a query that starts with a question word (who, what, when, where, why or how), be sure to add a space and some extraneous syntax such as a +, otherwise the natural language query heuristic will discard that trailing question mark. For example:

User Entry	Behavior
What is aspirin?	The question mark is ignored
myField: XX/YY/Z?	The question mark is treated as a wildcard
Where is part AB004x?	The question mark is ignored
Where is part AB004x? +	The question mark is treated as a wildcard and the extraneous "+" will be ignored
myField: XX/YY/Z? +	The question mark is treated as a wildcard and the extraneous "+" will be ignored

## Wildcards at Start of Terms

Wildcards can be placed at the start of terms, such as *\*ation*, which is known as a *leading wildcard* or sometimes as a *suffix query*. The syntaxes are the same as described above, but there may be local performance considerations that need to be evaluated.

Lucene and Solr technically support leading wildcards, but this feature is usually disabled by default in the traditional query parsers due to concerns about query performance since it tends to select a large percentage of indexed terms. The Lucid query parser does support leading wildcards by default, but this feature may be disabled by setting the `leadWild` configuration setting in `solrconfig.xml` to 'false'. To address performance concerns, Lucene 2.9+ and Solr 1.4+ now support a 'reversed wildcards' (or 'reversed tokens') strategy to work around this performance bottleneck.

This optimization is disabled by default. To enable this optimization you must manually add the `ReversedWildcardFilterFactory` filter to the end of the index analyzer tokenizer chain for the field types in the `schema.xml` file for the fields that require this optimization.

This affects all fields for the selected field types, so if you have multiple fields of a selected type and do not want this feature for all of them, you must create a new field type to use for the selected field.

The Lucid query parser will detect when leading wildcards are used and invoke the reversal filter, if present in the index analyzer, to reverse the wildcard term so that it will generate the proper query term that will match the reversed terms that are stored in the index for this field.

The rules for what constitutes a leading wildcard are not contained within the Lucid query parser itself. Rather, the query parser invokes the filter factory (if present) to inquire whether a given wildcard term satisfies the rules. There are a variety of optional parameters for the filter factory, described below, to control the rules. The default rules are that a query term will be considered to have a leading wildcard and to be a candidate for reversal only if there is either an asterisk in the first or second position or a question mark in the first position and neither of the last two positions are a wildcard. If a wildcard query term does not meet these conditions, the wildcard query will be performed with the usual, un-reversed wildcard term.

Use of the wildcard reversal filter will double the number of terms stored in the index for all fields of the selected field type since the filter stores the original term and the reversed form of the term at the same position.

There is no change to the query analyzer for the optimized field or field type. The reversal filter factory must only be specified for the index analyzer.

As an example, the index analyzer for field type `text_en` should appear as follows after you have manually edited `schema.xml` to add the wildcard reversal filter at the end of the index analyzer for this field type:

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

You must place the wildcard reversal filter at the end of the index analyzer for the field type since it is reversing the final form of the terms as they would normally be stored in the index.

Although this feature improves the performance of leading wildcards, it will not improve the performance of search terms that have both leading and trailing wildcards, since such a term will still have a leading wildcard even after being reversed. In such a case, which depends on the rule settings, the filter factory will inform the Lucid query parser that such a wildcard term is not a candidate for reversal. In that case, the Lucid query parser would generate a wildcard query using the un-reversed wildcard term.

The filter factory has several optional parameters to precisely control what forms of wildcard are considered leading and candidates for reversal at query time:

- `maxPosAsterisk="n"` -- maximum position (1-based) of the asterisk wildcard ('\*') that triggers the reversal of a query term. Asterisks that occur at higher positions will not cause the reversal of the query term. The default is 2, meaning that asterisks in positions 1 and 2 will cause a reversal (assuming that the other conditions are met.)
- `maxPosQuestion="n"` -- maximum position (1-based) of the question mark wildcard ('?') that triggers the reversal of a query term. The default is 1. Set this to 0 and set `maxPosAsterisk` to 1 to reverse only pure suffix queries (i.e., those with a single leading asterisk.)

- `maxFractionAsterisk="n.m"` -- additional parameter that triggers the reversal if the position of at least one asterisk ('\*') is at less than this fraction (0.0 to 1.0) of the query term length. The default is 0.0 (disabled.)
- `minTrailing="n"` -- minimum number of trailing characters in query term after the last wildcard character. For best performance this should be set to a value larger than one. The default is two.

These optional parameters only affect query processing, but must be associated with the index analyzer even though they do not affect indexing itself.



## Range Queries

A range query is a pair of terms which matches all terms which are lexically between those two terms. The two terms are enclosed within square brackets ("[]") or curly braces ("{}") and separated by the keyword "TO". For example,

- [cat TO dog]

Square brackets indicate that the specific terms will also match terms in documents. This is referred to as an inclusive range. Curly braces indicate that the specific terms will not match terms in documents and that only terms between the two will match. This is referred to as an exclusive range.

LucidWorks supports open-ended ranges, in which one or both terms are written as an asterisk ("\*") to indicate there is no minimum or maximum value for that term.

In contrast to Lucene and Solr, the brackets and braces can be mixed in LucidWorks, so that one term (either) can be inclusive while the other is exclusive.

Another LucidWorks extension allows the "TO" keyword to be entered in the lower case ("to"). Some examples:

User input	Matches
[cat TO dog]	All terms lexically between "cat" and "dog", including "cat" and "dog"
{cat TO dog}	All terms lexically between "cat" and "dog", excluding "cat" and "dog"
{cat TO dog]	All terms lexically between "cat" and "dog", excluding "cat", but including "dog"
[cat TO dog}	All terms lexically between "cat" and "dog", including "cat", but excluding "dog"
[* to dog]	"dog" and all terms lexically less
[cat to *]	"cat" and all terms lexically greater.
[cat to *}	Same as above because "*" forces inclusive match.
[* to *]	All terms
{* to *}	Same, because "*" forces inclusive match.

Range queries only work properly for fields that are lexically sorted or trie numeric fields, as specified by the administrator.

## Trie Numeric Range Query

Lucene now supports *trie numeric* fields, which enable much faster range queries in addition to being sorted in numeric order rather than lexical order. Each trie numeric field has a datatype, integer, real, or date.

User input	Matches
pageCount:[10 to 20]	Documents between 10 and 20 pages in size
weight:[0.5 to 70]	Documents with weights between 0.5 and 70.0
dateModified:[1984 to 2005]	Documents modified between 1984 and 2005
dateModified:[20-Jan-04 to 05-Feb-07]	Documents modified between January 20, 2004 and February 5, 2007

## Fuzzy Queries

A fuzzy query is a request to match terms that are reasonably similar to the query term. A wildcard is a strict form for specifying similarity, requiring the individual characters of the term to match precisely as written, while a fuzzy query allows characters to be shuffled, inserted, or deleted. Fuzzy query measures the editing distance, which is the number of characters which would have to be moved, inserted, or deleted to match the query term and then comparing that to half the length of the query term as a ratio. In other words, half as many characters as appear in the query term can be shuffled or changed in order to match the query term. Fuzzy query is good for matching similar terms as well as catching misspellings.

A fuzzy query is simply a single term (not a phrase) followed by a tilde ("~"). For example,

`soap~` matches "soap", "soup", "soan", "loap", "sap", "asoa", and so on.

For cases in which you want to require greater or less similarity, an optional similarity ratio can be specified.

The similarity ratio is a float ratio written after the tilde, ranging from 0.0 to 0.999. The default is 0.5. Smaller ratios indicate that less similarity is required. Larger ratios indicate that greater similarity is required. Lucene does not support a ratio of 1.0 which would require no differences from the query term. Lucid will treat any value of 1.0 or greater as 0.999, which effectively requires an exact match unless the term is very long.

Lucene's implementation of fuzzy search is not very effective for very short terms (three characters or less) because it uses half of the term length as the maximum editing distance.

## Fields and Field Types

An enterprise search engine organizes the data and properties for documents of the collection into distinct categories called *fields*, each of which has its own type of data as defined by a *field type*. Fields and field types are defined in a *schema*. The formatting and handling of data within fields is performed by a software component known as an *analyzer*. Analyzers are defined for each field type. There is a *crawler* which collects documents and data and analyzes their content using the analyzers and stores the analyzed data in the *index*, where it can be searched by user queries. This is an open-ended architecture that provides tremendous flexibility, but can also add to the level of detail that users, or at least advanced users, need to know to query a LucidWorks search collection.

The system administrator is responsible for setting up the schema and defining the fields and field types. Those details are beyond the scope of this section and are generally not needed by a typical user. The only information the user needs is the list of field names that are supported by the schema and possibly details about how fields may be formatted. Many fields will be simple text fields, so no additional detail is needed, and many other fields will be simple numbers or strings or dates. However, for sophisticated enterprise applications, there may be fields with more complex structure.

The LucidWorks query parser recognizes four distinct field types:

- Text
- Date
- Numeric (trie)
- Other

Text fields behave virtually the same as in a typical Web search engine, with queries consisting of words and quoted phrases.

Date fields have a standard format, but because dates formats vary so widely, a variety of alternative formats are recognized by the Lucid query parser to allow users to more easily specify dates in queries. For example, the year alone can be used in a query rather than writing a range from January 1 to December 31, or a range of years can be used without the non-year portion of the standard format. See [Date Queries](#) for more information.

For trie numeric fields, the Lucid query parser will make sure that numbers are properly formed before being handed off to the analyzer. For integer fields, real numbers used as query terms will be truncated to integers. Negative numbers are also permitted for trie numeric fields.

For all other field types, the Lucid query parser simply hands the source query term text off to the analyzer and relies on it to "do the right thing." If the analyzer has a problem, the term will be ignored.

## Related Topics

- [LucidWorks FieldTypes API](#)

- 
- LucidWorks Fields API

## Field Queries

When thinking about what is to be searched, the user has two choices: either to rely on the default field(s) specified by the system administrator or to explicitly specify field names within the query. In general, the former is sufficient, but on occasion the latter is needed.

Lucene had the concept of a single default field, but in Solr that has been replaced with a list of fields known as the *query fields* or *default fields*. Of course, there is no reason that the query fields list could not consist of a single field. Searching for a term will actually search for it in each of the fields listed in the query fields, and a document will match for that term if the term is found in any of the query fields.

Because every enterprise application has its own data needs, you will have to consult your system administrator for the list of field names.

There are three query formats for explicitly specifying field names:

- `title:aspirin`  
Single term - write the field name, a colon, and then the single term (or quoted phrase). Only that one term will be searched in the specified field and subsequent terms will be search in the default fields.
- `title:(cat OR (dog AND fish))`  
Parenthesized sub-query - write the field name, colon, left parenthesis, a sub-query (which may simply be a full query, ranging from a single term to a complex Boolean query with nested parentheses), and a right parenthesis.
- `title:cat OR (dog AND fish)`  
Sticky field name - write the field name, a colon, a space, and then the rest of the query. All subsequent terms will be searched in that *sticky field name*, until a new sticky field name is specified or a right parenthesis is reached at the current parenthesis level, for example.

There are two special pseudo-field names:

- **ALL:** - searches all fields defined in the schema or even defined dynamically, for example,  
`ALL:cat OR dog`
- **DEFAULT** - revert to searching the default query fields if in a sticky or parenthesized field name.  
`title:cat OR dog AND DEFAULT:fish`



### FYI

Field names are case insensitive, including **ALL** (All, all) and **DEFAULT** (Default and default).

**Note**

If the query parser encounters a field name that is not defined, it will be treated as a simple term. For example, `noField:foo` will be treated as the term list `noField foo`. This is done because it is not uncommon to paste natural language text into the search box, and colon is a not uncommon punctuation character.

## Date Queries

When the LucidWorks query parser detects that a field is defined in the collection schema as the field type "date" (or an instance of a "DateField", or a trie), a variety of common date formats are also supported in addition to the traditional Solr date format as defined in the "Solr Date Format" section. These alternative formats are designed to make it easy to specify individual dates, months, and years. They can be used as standalone query terms, or with the range and relational operators. The non-Solr, alternative, date formats will automatically be expanded internally by the LucidWorks query parser into Lucene range queries. For example, 2006 would be treated the same as [2006-01-01T00:00:00Z TO 2006-12-31T23:59:59.999Z].

The common date formats are:

- YYYY
  - 2001
- YYYY-MM
  - 2001-07
  - 2001-7
- YYYY-MM-DD
  - 2001-07-04
  - 2001-7-4
- YYYYMM
  - 200107
- YYYYMMDD
  - 20010704
- YYYY-MM-DDTHH
  - 2001-07-04T08
  - 2001-07-04t08
  - 2001-7-4t08
- YYYY-MM-DDTHH:MM
  - 2001-07-04T08:30
  - 2001-07-04t08:30
  - 2001-7-4t08:30
- MM-YY
  - 07-01
  - 7-98
  - 7-1
- MM-YYYY
  - 07-2001
  - 7-2001
- MM-DD-YY
  - 07-04-01
  - 7-4-01
  - 7-4-1
- MM-DD-YYYY



- 07-04-2001
- 7-4-2001
- MM/YY
  - 07/01
  - 7/01
  - 7/1
- MM/YYYY
  - 07/2001
  - 7/2001
- MM/DD/YY
  - 07/04/01
  - 7/4/1
- MM/DD/YYYY
  - 07/04/2001
  - 7/4/2001
- DD-MMM-YY
  - 04-Jul-01
  - 4-jul-01
  - 4-JUL-01
  - 04-Jul-1
  - 4-jul-1
  - 4-JUL-1
- DD-MMM-YYYY
  - 04-Jul-2001
  - 4-jul-2001
  - 4-JUL-2001
- MMM-YY
  - Jan-01
  - jan-01
  - JAN-1
- MMM-YYYY
  - Jan-2001
  - jan-2001
  - JAN-2001

## Date Ranges

Even a simple date term can expand into a range, but the alternative date formats can be used in range terms as well.

User Input	Query Interpreted As
[2001 TO 2005]	[2001-01-01T00:00:00Z TO 2005-12-31T23:59:59.999]
{2001 TO 2005}	{2001-12-31T23:59:59.999Z TO 2005-01-01T00:00:00Z}
[Jun-2001 to 7/2005]	[2001-06-01T00:00:00Z TO 2005-07-31T23:59:59.999Z]
[7/4/2001 to 9-7-2002]	[2001-07-04T00:00:00Z TO 2002-09-07T23:59:59.999Z]

The alternative data formats may also be used with relational operators to imply ranges.

User Input	Query Interpreted As
dateModified: < 2001	dateModified: [* TO 2001-01-01T00:00:00Z]
dateModified: <= 2001	dateModified: [* TO 2001-12-31T23:59:59.999Z]



Date format expansion only occurs for fields that the LucidWorks query parser recognizes as being "date" fields. A date stored in a text field or string field must be manually and correctly formatted by the user.

## Solr Date Format

The user-friendly date formats supported by the LucidWorks query parser are in addition to the date format that is supported by Solr. Solr provides a format for a specific date and time, as well as *date math* to reference relative dates and times based on a starting date and time.

The simplest form of Solr date is the keyword 'NOW' which refers to the current date and time. It is case sensitive in Solr, but the Lucid query parser will permit it to be in any case. 'NOW' can be used either if no explicit date or date math is specified, or it can be used if date math is specified without an explicit date.

An explicit date is written in Solr using a format based on ISO 8601, which consists of a string of the form `yyyy-mm-ddThh:mm:ss.mmmZ`, where 'yyyy' is the four-digit year, the first 'mm' is the two-digit month, 'dd' is the two-digit day, 'T' is the mandatory literal letter 'T' to indicate that time follows, 'hh' is the two-digit hours ('00' to '23'), the second 'mm' is the two-digit minutes ('00' to '59'), 'ss' is the two-digit seconds ('00' to '59'), optionally '.mmm' is the three-digit milliseconds preceded by a period, and 'Z' is the mandatory literal letter 'Z' to indicate that the time is UTC ('Zulu'). The millisecond portion, including its leading period, is optional. Trailing zeros are not required for milliseconds.

### Note

The Lucid query parser does not require the 'Z' and will translate a lower case 't' or 'z' to upper case.

Some examples:

- NOW
- Now
- now
- 2008-07-04T13:45:04Z
- 2008-07-04T13:45:04.123Z
- 2008-07-04T13:45:04.5Z

Solr requires hyphens between the year, month, and day, but the LucidWorks query parser will add them if they are missing.

Solr date math consists of a sequence of one or more addition, subtraction, and rounding clauses. '+' introduces an addition clause, '-' introduces a subtraction clause, and '/' introduces a rounding clause. Addition and subtraction require an integer followed by a calendar unit. Rounding simply requires a calendar unit.

The calendar units are:

- YEAR or YEARS
- MONTH or MONTHS
- DAY or DAYS or DATE
- HOUR or HOURS
- MINUTE or MINUTES
- SECOND or SECONDS
- MILLISECOND or MILLISECONDS or MILLI or MILLIS

Example rounding clauses:

- /YEAR
- /MONTH
- /DAY
- /HOUR

Example addition and subtraction clauses:

- +6MONTHS
- +3DAYS
- -2YEARS
- -1DAY

Examples using dates and date math:

- NOW
- NOW/DAY
- NOW/HOUR
- NOW-1YEAR
- NOW-2YEARS
- NOW-3HOURS-30MINUTES

- /DAY
- /HOUR
- -1YEAR
- -2YEARS
- /DAY+6MONTHS+3DAYS
- +6MONTHS+3DAYS/DAY
- 2008-07-04T13:45:04Z/DAY
- 2008-07-04T13:45:04Z-5YEARS
- [2008-07-04T13:45:04Z TO 2008-07-04T13:45:04Z+2MONTHS+7DAYS]



Whitespace is not permitted.

As a general rule, any tail portion of a proper date/time term can be omitted and the Lucid query parser will fill in the missing portions. But, the result will be an implicit range query. For example:

- 2008-01-01T00:00:00Z
- 2008-01-01T00:00:00
- 2008-01-01T00:00 **same as** [2008-01-01T00:00:00Z TO 2008-01-01T00:00:59Z]
- 2008-01-01T00: **same as** [2008-01-01T00:00:00Z TO 2008-01-01T00:59:59Z]
- 2008-01-01T **same as** [2008-01-01T00:00:00Z TO 2008-01-01T23:59:59Z]
- 2008-01-01 **same as** [2008-01-01T00:00:00Z TO 2008-01-01T23:59:59Z]
- 2008-01 **same as** [2008-01-01T00:00:00Z TO 2008-01-31T23:59:59Z]
- 2008 **same as** [2008-01-01T00:00:00Z TO 2008-12-31T23:59:59Z]

The same technique can be used in explicit date range queries and with relational operators.

See the `org.apache.solr.schema.DateField` and `org.apache.solr.util.DateMathParser` Java classes for more information about the Solr date/time format.

If there is any parsing problem in a date term, the LucidWorks query parser will catch the exception and simply ignore the date term.

## Non-Text, Date, Numeric Field Queries

The LucidWorks query parser has a number of features to support text, date, and trie numeric fields, but other field types are simply passed through the schema query analyzer that has been specified by the administrator in the schema. Non-trie numbers (integers, floats) and strings are two common non-text/date/numeric field types.

String fields may seem similar to text fields, but the full string is one term, even if it has embedded whitespace and punctuation.

If you make a mistake in formatting query text for a non-text, non-date, non-trie numeric field, the offending query term will simply be ignored. String fields would not typically have any mistakes, but any mistakes could cause a failure to match documents properly. A non-digit or a decimal point in an integer field or an improperly formatted float are mistakes that could occur in non-trie numeric fields.

There are two forms for terms in non-text/date/numeric fields:

- **Simple term:** a term as in a text field, typically delimited by whitespace or one of the special syntax characters.
- **Quoted string:** looks like a quoted text phrase, but every character between the quotes, including whitespace and any special syntax characters are considered part of the text of the term.

In both forms any special syntax characters or whitespace can be escaped (with backslash). For many cases, either form can be used. An advantage of quoted string terms is that less escaping of special syntax characters is needed. A key difference is that wildcard character cannot be used as wildcards within quoted strings, but they can be used as wildcards in simple terms.

Some examples:

- `myIntField:123`
- `myFloatField:123.45`
- `myStringField:Hello`
- `myStringField:Hello\ World!` (With escaped embedded space and exclamation point.)
- `myStringField:"Hello World!"` (Same, but no escaping needed.)
- `myIntField:"123"`
- `myStringField:*cat*` (Wildcard matches string values containing the sub-string "cat".)
- `myStringField:"cat"` (Non-wildcard matches Matches string values that are literally "**cat**".)
- `myStringField:"\"cat\""` (Matches string values that are the text "cat" with enclosing double quotes.)
- `myStringField: ({[A~B:C^D]})` (Matches the string "({[A~B:C^D]}").)
- `myStringField:"({[A~B:C^D]}")` (Same, but no escaping needed.)

## Whitespace

Whitespace can be used as liberally as a user desires between terms and operators, with only a few exceptions. There must not be any space:

- Between a field name and the colon (':') which follows it.
- Between a term operator ('{+}' or '-') and the term that follows.
- Between a term and a suffix modifier (tilde or circumflex), or within a suffix modifier, or between suffix modifiers.
- Between a backslash ("\") and the special character that it is escaping.

Also, whitespace is *not* required, but permitted at the following points:

- Before or after parentheses.
- Before or after a non-keyword Boolean operator ("&&", "||" or "!").
- Before or after a double quotation mark (") enclosing phrases.
- Before a term operator ('{+}' or '-').

There must be either whitespace or some operator, such as a parenthesis, after any single term and any '+' or '-' operator that follows it, otherwise the '+' or '-', and any non-whitespace that follows, is considered as part of the preceding term.

Line breaks are treated as whitespace. So, very long queries can be broken into shorter lines for readability with no impact of the query interpretation.

Quoted phrases may be split over two or more lines as well.

## Term Operators

In addition to the Boolean operators ('AND', 'OR', 'NOT') used to construct complex queries with sub-queries, there are operators that are used at the term level, within term lists. They are written immediately before a term, without whitespace.

- '+' - Require a term - it must be present in documents
- '-' - Exclude a term - it must not be present in documents
- Relational operators - '==', '!=', '<', '<=', '>', and '>='. Whitespace is permitted after the operator.

Term operators, field names, keyword options, and prefix modifiers can be written in any order. For example:

```
title:>cat
```

```
>title:cat
```

```
+title:dog
```

```
title:+dog
```

```
-title:frog
```

```
title:==frog
```

```
==title:frog
```

```
nostem:+title:bats
```



## Selecting All Documents

The LucidWorks query parser has a special feature that selects all documents as efficiently as possible using a special Lucene API. The syntax is simply `*:*`. The LucidWorks default [Search UI](#) allows querying for all documents simply by hitting the **Search** button with an empty query box.

Alternately, a query consisting of a single asterisk (`'*`') will select all documents that contain a term in the list of default search fields. This will not necessarily select all documents since there may be some documents in the collection which do not have values in the default query fields.

All documents containing a term in a specified field can be queried by writing the field name, a colon, and the asterisk. For example,

- `title:*`

would return all documents that have titles, although not all documents may have titles.

Combining that with exclusion permits a query to find all documents that do not have a value in the specified field. For example,

- `-title:*`

is equivalent to:

- `*:* -title:*`

But that is not necessarily equivalent to:

- `* -title:*`

because the latter depends on precisely which fields are listed in the query fields, so it may return no documents or a subset of the total documents in the collection.

## Relational Operators

In addition to the Lucene range syntax, the LucidWorks query parser supports the standard collection of *relational operators* (also known as *comparative operators*, '=', '!=', '<', '<=', '>=', and '>'), for example:

- Nevada politics >= dateCreated: 2003
- Nevada politics dateCreated: >= 2003

Whitespace may be freely used both before and after a relational operator.

Other term prefix modifiers, such as a field name and term keyword options, can be combined with a relational operator, in any order. For example, the following queries are equivalent:

- cat nosyn:body: < dog
- cat < nosyn:body: dog
- cat nosyn:body: <dog
- cat nosyn: <body: dog
- cat nosyn: < body: dog
- cat nosyn: body: < dog
- cat nosyn: body: <dog

The '=' and '!=' relational operators are equivalent to the '+' and '-' term operators. So, these queries are equivalent:

- cat +dog -fox
- cat ==dog !=fox
- cat == dog != fox

## Accented Characters

The LucidWorks query parser supports text terms written using the so-called accented characters that appear in Unicode and ISO Latin-1 as hex codes 00A1 through 00AF, but it is common that the administrator will set up the schema so that a *filter* such as the `ISOLatin1AccentFilter` will be included in field type analyzers for the purpose of *stripping* the accents by mapping the accented characters to unaccented ASCII equivalents.

For example, `Café Française` would be treated identically to `Cafe Francaise`

Typically, accents are removed when documents are indexed. That means that they must also be removed at query time so that query terms can match indexed terms. But, the administrator could decide to preserve accented characters at index time, in which case accents will then also need to be preserved at query time.

The LucidWorks query parser normally bypasses the analyzer for text fields, but it will invoke the accent removal filter if it is present and has the word `Accent` in its name.

It is technically possible for the administrator to construct an index filter that indexes both the accented and unaccented forms of terms and remove the query accent filter, and then the user could query the unaccented term to get both accented and unaccented terms or query the accented term and get only the accented terms.

Accents are not normally stripped for non-text fields, but that depends purely on whether each non-text field type does or does not have an accent removal filter specified.

## Building Advanced Queries

---

This section describes more advanced search queries some of the most commonly used types of search queries and gives examples of how LucidWorks processes them.

## Minimum Match for Simple Queries

None of the optional terms in a simple Boolean query is required to be present for a document to be selected by the query, but in some cases you would like to require that at least *some* of the optional terms be present. This can be accomplished by supplying a *minimum match* modifier, which specifies either a count or percentage of the optional terms that are required for the immediate following simple Boolean query. The minimum match can be specified either with the `minMatch` (or `atLeast`) keyword option or by enclosing the simple query within parentheses and appending a tilde ('~') followed by the term count or percentage (which may also be a fraction). The keywords `minMatch` and `atLeast` are synonymous. For example:

- `minMatch:1 +pet cat dog fish rabbit -snakes`
- `minMatch:2(+pet cat dog fish rabbit -snakes)`
- `minMatch:25%(+pet cat dog fish rabbit -snakes)`
- `minmatch:0.25(+pet cat dog fish rabbit -snakes)`
- `minMatch:50% +pet cat dog fish rabbit -snakes`
- `minmatch:25(+pet cat dog fish rabbit -snakes)`
- `atLeast:25(+pet cat dog fish rabbit -snakes)`
- `atleast:25(+pet cat dog fish rabbit -snakes)`
- `(+pet cat dog fish rabbit -snakes)~1`
- `(+pet cat dog fish rabbit -snakes)~25%`
- `(+pet cat dog fish rabbit -snakes)~25`
- `(+pet cat dog fish rabbit -snakes)~0.25`

If a space follows the `minMatch` keyword option, then the setting is *sticky* and applies to all subsequent Boolean queries until the next closing parenthesis, otherwise the setting applies only to the parenthesized Boolean query that immediately follows.

Since each of the above examples has four optional terms, 25% means that one out of four of the optional terms must be present in a document for it to be selected by the query. A value of 50% (or two) requires that at least half of the optional terms be present in a document.

A value of 0 or 0% means that no optional terms are required. This is the default. A value that matches (or exceeds) the count of optional terms or 100% means that all optional terms are required.

If a whole number is specified and no percentage is present, the Lucid query parser will do an excellent job of *guessing* whether the number is a count of terms or a percentage. In other words, the percent symbol ('%') is almost always optional.

As a special case, a small percentage, such as 1%, is treated as requiring a minimum of one optional term to match.

Keyword option names are *not* case sensitive, although they tend to be written in their proper *camel case* form in this documentation. So, `minMatch`, `minmatch`, `MinMatch`, and `MINMATCH` are all equivalent.

The administrator can change the default (for example, to one to assure that at least one optional term is present) using the `minMatch` configuration setting.

## Negative Queries

A term list with only the '-' term operator and no '+' term operators is known as a *negative query* and will query all documents that do not have the specified '-' terms. This is equivalent to a term list requesting all documents and then excluding the specified terms.

User Input	Equivalent to
-cat	*:* -cat
-cat -dog	*:* -cat -dog

## Escaping Wildcard Characters

The wildcard characters, "\*" and "?" are a special case. They are always part of the term in which they are embedded, but they have their special wildcard meaning rather than being simply characters in a term. But, if you do have a non-text field in which the wildcard characters are actually text in that field, you can escape them using a backslash. For example,

- `myField: E\*TRADE`

The term will literally be "E\*TRADE" rather than a wildcard.

- `myField: x\?y\?z`

The term will literally be "x?y?z" rather than a wildcard.

Note that due to a limitation of Lucene, if there are any non-escaped wildcard characters in a term, escaping will be ignored for all other wildcard characters in that term. For example,

- `myField: E\*TRA?E`

Will be treated as a wildcard query for the term "E\*TRA?E", with both the \* and ? being treated as wildcard characters. On the other hand,

- `myField: E\*TRA\E`

Will be treated as a non-wildcard term query for the term "E&\*TRA?E".



## Proximity Operations

A *proximity* query searches for terms that are either near each other or occur in a specified order in a document rather than simply whether they occur in a document or not.

### Phrase Proximity Queries

Exact phrase matching is a powerful query tool, but frequently the phrasing used in relevant documents is not exactly the same. It is commonly the case that there are extra terms, or the terms may be in another order. In other cases, the phrase terms may be relatively near, with quite a few extra words between them. For example, the following two queries may return different results even though they are semantically equivalent:

```
"team development"
```

```
"development of teams"
```

The difference between the two is an extra word in the middle and a reversal of the two key terms.

We can write a single *phrase proximity query* that will match both phrases:

```
"team development"~3
```

The tilde ("~") is used after a quoted phrase to indicate a phrase proximity search. It is followed by an integer (whole number) which is the maximum editing distance for phrases that will match the query phrase. The editing distance treats each term as a single unit and measures how many unit terms need to be moved to translate from one phrase to another. In this case, it takes one unit to move "team" to "of", a second unit to move it to "development", and a third unit to move it before "development".

To query for two terms that are within 50 words of each other:

```
"cat dog"~50
```

To query a person's name and allow for an optional middle initial:

```
"John Doe"~1 matches "John Doe" and "John Q. Doe"
```

To query a person's name and allow for both first name first or last name first:

```
"John Doe"~2 matches "John Doe" and "Doe, John", as well as "John Q. Doe"
```

### Advanced Proximity Operators

The Lucid query parser also supports advanced proximity query operators to specify more elaborate sequences of terms and to control the order of terms and how many intervening terms are permitted. The advanced proximity operator keywords are:

Advanced Operator	Sample Query	Matches
NEAR	x near y	Documents containing "x" within 15 terms of "y", either before or after
BEFORE	x before y	Documents containing the term "x" no more than 15 terms before the term "y"
AFTER	x after y	Documents containing the term "x" no more than 15 terms before the term "y"

These operators are case insensitive and may be upper, lower, or mixed case, unless the `opUp` configuration setting is set to "true", which would then treat them (and all other operator keywords) as normal terms unless they are entirely upper case.

### Excluding Terms from Advanced Proximity Queries

Normally, any combination of terms may appear between the terms that mark the start and end of an advanced proximity query (the BEFORE, AFTER, and NEAR operators), but in some situations it is desirable to prevent specific terms from occurring between those start and end terms. Just as with a simple keyword query, this exclusion can be done by listing terms preceded by the minus sign '-' or NOT operator.

For example, these pairs of queries are equivalent:

```
George NEAR Washington -person
George NEAR Washington NOT person
```

```
George NEAR Lincoln -person
George NEAR Lincoln (NOT person)
```

Also, the exclusions may be specified on either side of the proximity operator, so the following queries are equivalent:

```
George NEAR Lincoln -person
George -person NEAR Lincoln
```

```
George NEAR Lincoln (NOT person)
George (NOT person) NEAR Lincoln
```

### Controlling Distance Between Terms

By default, the distance between the two terms of a proximity operator can be up to 15 additional terms. That default distance is controlled by the `nearSlop` configuration setting. But if you need

more or fewer intervening terms for a specific proximity operator, you can specify the desired limit of intervening terms by writing a colon (":") and the number immediately after the operator name. For example,

```
x before:3 y
```

matches documents containing "x" no more than three terms before "y".

A distance of 0 (zero) means no intervening terms. For example,

```
x before:0 y
```

is the same as:

```
"x y"
```

which matches documents where the terms are adjacent and in that order.

## Composing Longer Sequences of Terms

The advanced proximity operators can be composed (or "daisy-chained") to match more complex term sequences. For example:

```
x before y before z
```

matches documents containing "x" before "y" with no more than 15 intervening terms and followed by "z" with no more than 15 intervening terms after "y".

The distance limit can be controlled for each proximity operator, such as:

```
x before:10 y before:100 z
```

which requires that there be no more than 10 terms between "x" and "y", but "z" can be up to 100 terms after "y".

Any combination of any number of NEAR, BEFORE, and AFTER proximity operators can be composed into a sequence, such as

```
cat near dog before:50 fox after fish near:3 bat before zebra
```

## Left to Right Evaluation Order

When multiple advanced proximity operators are composed, they are evaluated left to right, except as parentheses are used to explicitly specify the evaluation order. So, the previous example is evaluated as:

```
(x before y) before z
```

In fact, the evaluation order does not matter in that example, which could also be written as:

`x before (y before z)`

But evaluation order does matter with:

`x near:3 (y before:50 z)`

where the intent is that "x" could be shortly before or after either end of the "y"/"z" sequence. But,

`x near:3 y before:50 z`

would evaluate as:

`(x near:3 y) before:50 z`

which would match "x" close to "y" but not close to "z".

Within parentheses used for operands of proximity operators, only the OR and proximity operators can be used. Other operators will be treated as if they were the OR operator.

## Quoted Phrases

Quoted phrases with any number of terms can be used as the operands of the proximity operators. For example,

`"First step" before:200 "last step"`

The terms in the quoted phrase must occur in order, with no intervening terms between the quoted terms.

## Quoted Proximity Phrases

Quoted phrases may specify a maximum number of terms that may appear between the terms of the phrase, using the usual quoted phrase proximity query notation of a tilde ("~") and the number of terms permitted. For example,

`"proposal development"~3 near:50project`

Would match the terms "proposal" and "development" (in that order) with no more than three intervening terms and occurring no more than 50 terms before or after "project".

### Note

Unlike normal quoted proximity phrases, the phrase terms are expected to occur in order. So, this example will not match `"development proposal...project"`.

## Alternative Terms

When several different terms are permitted at a position in a proximity sequence, the alternative terms can be specified using the OR operator and parentheses for either or both terms of the operator. For example,

```
(cd-rom or dvd) before:1 drive
```

would match documents with the term "drive" preceded by either "cd-rom" or "dvd" with at most one intervening term. Alternatives can also be used with composed proximity operators. For example,

```
(cd-rom or dvd) before:1 ((built-in or external) before:0 drive)
```

which requires "built-in" or "external" to immediately precede "drive", but an intervening term is permitted after "cd-rom" or "dvd".

Alternatives can also be quoted phrases. For example, ("In the beginning" or "At the start" or "Starting out") before:1000 "the end" will match documents containing the phrase "the end" preceded by either the phrase "In the beginning", "At the start", or "Starting out" with up to 1,000 intervening terms.

## Term Lists

A phrase that is not enclosed within quotes is known as a *term list* and may be used as either of the operands of a proximity operator, where it will be treated as if it were a quoted phrase. For example,

```
pets before animal judgments before book
```

will match the same documents as:

```
pets before "animal judgments" before book
```

## Parenthesized Proximity Expressions in Term Lists

Although term lists with proximity operators may seem like a mere convenience to avoid typing the quotes around a phrase, the construct is much more powerful. Each of the terms in a proximity term list can be one of:

- Single term (but no wildcard or fuzzy term)
- Quoted phrase
- Parentheses enclosing:
  - One or more proximity operators (evaluated left to right)
  - Another term list
  - List of term alternatives separated by OR operators. Each term alternative can be a full proximity expression, including nested parentheses.

For example,

```
red (light or sign) picture near street
```

would be equivalent to:

```
("red light picture" or "red sign picture") near street
```

which could also be written using nested term lists:

```
(red light picture or red sign picture) near street
```

which is also equivalent to:

```
((red light picture) or (red sign picture)) near street
```

## Single Field

Although field names can be used for terms within a proximity expression, only the first field name is used and the others are ignored since an entire proximity expression is evaluated within only a single field.

```
title:x after (body:y near author:z)
```

is evaluated as:

```
title: x after (y near z)
```

A proximity query with no field or the DEFAULT field will query against all of the fields listed in the `qf` (query fields) request parameter. The proximity query will be evaluated against each field in turn and the results combined with the disjunction max query operation. But, that will still evaluate the full proximity query expression on only one field at a time.

## Boolean Operations on Proximity Expressions

Multiple proximity expressions, each with its own field, can be used within a single query simply by combining them with the AND, OR, or NOT boolean operators. The precedence of the boolean operators is such that entire proximity expressions will be evaluated before the surrounding boolean operators. So,

```
title: red before light or body: empty before tank
```

would evaluate as:

```
(title: red before light) or (body: empty before tank)
```

The AND operator can be used to require a set of proximity queries to be satisfied, such as:

```
(title: red before blue) and (body: night after day) and (town near city)
```

---

where "red" must occur before "blue" in the title field, "night" must occur after "day" in the body field, and "town" must occur near "city" in any field.

## Term Boosting

Although the Lucid query parser will automatically add relevancy boosting for bigrams and trigrams of query terms, the sophisticated user may also explicitly add a boost factor for any term.

A boost factor is a suffix modifier placed after a term which consists of a circumflex ('^') followed by a decimal number indicating a multiplication factor to use in the relevancy calculation for a term. For example:

- `cat^4 dog^1.5 fox "the end"^0.3`

The default boost factor is 1.0, but it is actually derived from the default boost factors specified for the various fields given in the default query field configuration which is controlled by the administrator.

A boost factor of 1.0 indicates that there should be no change from the default boost factor. A factor greater than 1.0 will increase the relevancy of the term. A factor less than 1.0 will decrease the relevancy of the term.

In the example above, "fox" will get the default boosting, "dog" will get modestly higher boosting, "cat" will get significantly higher boosting, and "the end" will have its relevancy reduced well below the default.

You can also give a relevancy boost factor to a term list or sub-query by enclosing it within parentheses. For example:

- `(cat +dog -fox)^2.5`
- `(cat OR dog)^3.5 AND (fox NOT bat)^0.5`



## Boolean Relevancy Boosting

Boolean 'AND' and 'OR' operators will also participate in relevancy boosting, by treating the operators as the text words 'and' and 'or' and then combining them into phrases with the last term of the term list to their left and the first term of the term list to their right. For example, each of the following queries will give a higher relevancy ranking for the phrase 'hit and run' than a document that simply contains the terms 'hit' as well as 'run':

- hit AND run
- hit and run
- hit && run
- hit "and" run
- "hit and run"

## Query Analysis for Relevancy Boosting

Bigrams, trigrams, unigrams, and n-grams are generated in the analysis of a user query to identify sequences of terms; in this context, an n-gram is a series of terms or words, though in other contexts an n-gram may refer to a series of characters. A bigram is a sequence of two terms, a trigram is a sequence of three terms, a unigram is a single term, and an n-gram is any sequence of terms, but generally four or more terms. A term list is an n-gram. In the context of n-grams, a quoted phrase counts as a single term. Although the user of the Lucid query parser does not need to worry about this level of detail since it is handled automatically by LucidWorks, it is helpful to understand how Lucid is going to analyze the query, perform the search, and rank the results for better relevance. Basically, the Lucid query parser uses the n-grams (particularly the bigrams and trigrams) from the original query to boost results that contain not only the discrete query terms, but n-grams of the query terms as well.

For example, both of the following queries match documents containing the phrase "meet the press":

- `meet the press`
- `"meet the press"`

The first query consists of three terms, or two bigrams ("meet the" and "the press"), and one trigram ("meet the press"). The second is actually a unigram because a quoted phrase counts as a single term.

The first query also returns any documents that contain "meet" and "press" ("the" is a stop word), even if the document does not contain the full sequence "meet the press". Traditionally that might be annoying, but the Lucid query parser automatically adds extra clauses to the user query to `OR` in the bigrams and trigrams from the query with a boost factor so that occurrences of "meet the", "the press", and "meet the press" will rank higher than documents that merely contain "meet" and "the".

Superficially, the second query might seem better because it is more precise, but sometimes extra words may be present so that multi-word fragments of the phrase might match, but will not if the full phrase is used.

A simple natural language phrase can be used directly as a query and can be expected to return quite good results without the need to add extra operators or specific formats. This point may not be completely obvious when using a simple three-word phrase, but should be more clear with a longer sentence fragment, such as:

- `The company meeting was attended by employees`
- `"The company meeting was attended by employees"`

The precise quoted phrase will match the exact phrasing so will not catch the statement if it is reworded as "Employees attended the company meeting." But the first query will match that rewording, and rank it reasonably high due to the match on the trigram "The company meeting".

## Term Modifiers

One method for selecting advanced search features is the use of *term modifiers*, which precede or follow a term.

There are three forms of term modifier that may appear before a term, referred to as a *prefix modifier*, all of which consist of a name followed by a colon:

- **Field name**, for example:  
`title:cat`
- **Pseudo-field name**: ALL, DEFAULT, \*, for example:  
`ALL:cat DEFAULT:dog`
- **Keyword options**: nostem, nosyn, and so on. Any number of keyword options can be specified for a single term, each with its own colon, for example:  
`nostem:nosyn:title:paintings`



### Note

There is no whitespace between prefix modifiers or the term to which they apply, if they are intended to apply to the single term (or quoted phrase or parenthesized sub-query) immediately following the colon. But, if there is a space, then the modifier will be a *sticky modifier* that applies to all subsequent terms at the same parentheses nesting level.

A term may also be preceded by a relational operator or a '+' or '-' operator, but those are considered *term operators* rather than term modifiers.

There are three forms of term modifier that may appear after a term, also called a *suffix modifier*:

- **Boost factor**: a circumflex ('^') followed by a decimal number indicating a multiplication factor to use in the relevancy calculation for a term, which may be greater than 1.0 to increase boosting or less than 1.0 to lower boosting, for example:  
`cat^4 dog^1.5 "the end"^0.3`
- **Proximity distance**: a phrase followed by a tilde ('~') followed by an integer to specify that additional words may occur between the terms of the phrase, as well as reordering of the terms, up to the specified distance.  
`"product security"~5`
- **Fuzzy search**: a single term followed by a tilde ('~'), optionally followed by a decimal number to specify that the term should match similar terms, where the number specifies how similar.  
`soap~` (Defaults to 0.5.)  
`soap~0.5`  
`soap~0.1` (Not very similar.)  
`soap~0.99` (Virtually identical.)

**Note**

Lucene considers wildcards and range searches to be term modifiers, but in this guide they are discussed separately.

## Default Query Fields

Although explicit field names can be specified for all terms, a default set of field names will be used for terms which are not preceded by a field name (or sticky field name). The administrator must decide which fields make the most sense for default fields for the application. In some cases that may be a single field, which may be a merger of a number of separate fields, but it may also be a list of field names. The `qf` configuration setting lists the default fields and their boosts.

The default field list may also specify default term boost factors for the fields. A field name can be followed by a circumflex and the boost factor for that field.

For example, the administrator might set the default query fields configuration setting to:

```
body title^5 abstract
```

That will cause the query:

```
cat title:dog fox
```

to be equivalent to:

```
(body:cat title:cat^5 abstract:cat) title:dog^5 (body:fox title:fox^5 abstract:fox)
```

Technically, LucidWorks generates a *disjunction maximum query* if multiple fields are specified for the default query fields.

The LucidWorks query parser also has the ability to support an asterisk ("\*") for the field list to indicate that all fields should be searched when no field is specified for a query term.

## Empty Queries

The query user interface may prevent the user from entering an empty query, but the LucidWorks query parser supports an alternate query string (see the [q.alt configuration setting](#)) to be used in such cases. This string can be configured by the administrator, but defaults to `*:*`, which will return all documents.

## Queries with Unicode Characters

Although the LucidWorks query parser itself is capable of accepting Unicode characters directly, it is usually not very convenient to enter them on a typical computer keyboard. As with Lucene and Solr, LucidWorks supports a variation of the Java escaping format to enter explicit Unicode characters as hexadecimal character codes. Each explicit Unicode character is introduced with a backslash "\", the letter "u", followed by one to four hexadecimal digits.

Unlike Lucene and Solr, which require a lower-case "u" and require exactly four digits, LucidWorks allows upper-case "U" and any leading zero digits need not be entered, unless the first character after the hexadecimal code is itself a character which is used for a hexadecimal digit (digits "0" to "9", letters "a" to "f", and letters "A" to "F".) But as a general practice it is safest and most consistent to write the full four digits with any leading zeros

Examples for the word "Cat":

- `Ca\u0074`
- `Ca\U074`
- `Ca\u74`
- `C\u061t`
- `\u0043at` (zeroes needed since 'a' is a hex digit)
- `\u0043\u0061\u0074`

Examples for the word "Cattle9":

- `Cattle\u39`
- `Ca\u74tle9`
- `C\u0061\u061\u074\u074\u006ce9`
- `C\u0061\u74\u074\u6c\u65\u0039`
- `\u0043\u0061\u0074\u0074\u006c\u0065\u0039`

Explicit Unicode characters are assumed to be part of query terms and will not be interpreted as query operators. For example, `\u0021` will not be treated as if it were the "!" NOT operator.

## Escaping Special Syntax Characters

Many non-alphanumeric characters will be accepted within a term, such as hyphen (-), period (.), comma (,), asterisk (\*), at sign (@), number sign (#), dollar sign (\$), and semicolon (;), but a handful have special meaning to the Lucid query parser, such as the non-keyword Boolean operators parentheses ( ), colon (:), double quotation mark ("), circumflex (^), and tilde (~). Terms are commonly delimited with white space, but the special syntax characters will delimit terms as well.

The special syntax characters are:

```
&& (But a single "&" is in fact permitted in a term)
|| (But a single "|" is in fact permitted in a term)
\ (Backslash)
!
^
~
(
)
{
}
[
]
:
"
==
<
>
  (space)
```

A plus sign, "+", or minus sign, "-", at the beginning of a term is also treated as a special syntax character.

Usually, the user need not be concerned in any way about the special syntax characters unless they explicitly wish to use them as operators, but some non-text fields may have terms that use some of the special syntax characters. In those cases, individual special characters can be *escaped* by preceding them with a backslash, "\". Any character can be escaped without any harm or translation.

For example, all of the above listed special syntax characters can be escaped to be used in terms for non-text fields as shown in this odd-looking but valid query term:

```
myField: A\&\B\|\|C\D\!F^F~G\ (H)I\{J\}K\[L\M\ :N\"O\==P\<Q\>R
```

which passes the following term to the field analyzer for myField:



```
A&&B\| \| C\D\!F^F~G (H)I{J}K\[L\]M:N"O==P<Q>R
```

Alternatively, a term for a non-text field can simply be enclosed within quotes, although any quotes or backslashes within the term must still be escaped. The previous example can be written as follows:

```
myField: "A&&B\| \| C\D\!F^F~G (H)I{J}K\[L\]M:N\"O==P<Q>R"
```

Here are some examples of special syntax characters which do not need to be escaped, as per the rules given above:

- `http://www.foo.com/index.html`  
A URL
- `info@foo.com`  
@ has no special syntax meaning and periods are allowed
- `20010630T12:30:00Z`  
colon appears to be in a time value
- `AT&T`  
Single ampersand is considered a valid character in terms
- `ab|cd`  
Single vertical bar is considered a valid character in terms

The apostrophe or single quote mark, "'", is not treated the same as a double quotation mark. It is commonly used for contractions and possessives. It will be preserved for non-text fields, but the typical analyzer for text fields will discard it.

## Term Keyword Options

*Term keyword options* provide a flexible mechanism for controlling the interpretation of a term. A term keyword option is a keyword followed by a colon that appears before a term. Any number of keyword options can be specified for a single term, each with its own colon. For example:

```
nostem:nosyn:title:paintings
```

The supported term keyword options are:

- `like`: to indicate that specified terms are not required, but will boost relevancy if present, so that selected documents will be "like" the specified terms. Can also be used to select documents containing the most relevant terms from a document specified by its document id.
- `minMatch`, `atLeast`: to set the minimum count of percentage of optional terms in a term list that must be present in selected documents. See the "Minimum Match for Optional Terms of Simple Boolean Queries" section.
- `syn`, `nosyn`: to enable or disable synonym expansion of the following term.
- `stem`, `nostem`: to enable or disable stemming of the following term. Although supported by the parser, there is not currently search support for both stemmed and unstemmed terms.
- `debugLog`: to enable debug output for a query to permit the administrator to examine the detailed query interpretation.

Term keyword options and any field name can be written in any order, so that the following queries are equivalent:

```
nosyn:title:tv
```


```
title:nosyn: tv
```

## Like Term Keyword Option

You can use the `like` term keyword option to specify terms that are not required in documents but will enhance relevancy if they are present: this option selects documents that are *like* a set of terms rather than absolutely requiring the terms. This option can specify a single term, a parenthesized list of terms, or be written as a sticky option that applies to all subsequent terms at this current parenthesis level. For example,

- President like: Lincoln Washington
- President like:(Lincoln Washington)
- President like:Lincoln like:Washington

All three forms are equivalent and will return all documents that have the term "President", with documents that also contain "Lincoln" or "Washington" ranked higher, and documents containing all three ranked highest.

 The single-term form cannot be used if the term has any punctuation or digits, because that triggers the *like document* feature which extracts high-relevancy terms from the specified document and then uses that term list as if it were specified using the like term keyword option.

A query may not even have *any* required terms. For example,

- like: Lincoln Washington Roosevelt
- like:(Lincoln Washington Roosevelt)
- like:Lincoln like:Washington like:Roosevelt

All three forms are equivalent and will return all documents that have at least one of the terms "Lincoln", "Washington", or "Roosevelt", with documents containing more of the terms ranked higher.

The `like` option can be used to reference documents that have text *similar* to a passage. For example,

- like: Four score "and" seven years ago our fathers brought forth
- like:(Four score "and" seven years ago our fathers brought forth)

Both forms are equivalent and will return all documents that have at least one of the words listed, with documents containing more of the words ranked higher and with documents containing more of the words adjacent as listed ranked even higher. Note: The quotes around "and" are needed to prevent it from being interpreted as a boolean operator.

For some simple cases it may be more convenient to use the "+" operator. For example:

- +cat white stray

- `cat like:(white stray)`

Both forms are equivalent and will return documents that must have "cat", but with any documents also containing "white" or "stray" ranked higher. Note: If there are not explicit " + " or " - " operators, the query terms will all be treated as if "" were written.

The `like` option can also be used in conjunction with the `minMatch` option to require at least a specified fraction of the optional terms to be present in documents. For example,

- `minMatch:75% like: Four score "and" seven years ago our fathers brought forth`
- `minMatch:75 like: Four score "and" seven years ago our fathers brought forth`
- `minMatch:0.75 like: Four score "and" seven years ago our fathers brought forth`
- `minMatch:8 like: Four score "and" seven years ago our fathers brought forth`
- `minMatch:75%:(like: Four score "and" seven years ago our fathers brought forth)`
- `like:(Four score "and" seven years ago our fathers brought forth)~8`
- `like:(Four score "and" seven years ago our fathers brought forth)~75`
- `like:(Four score "and" seven years ago our fathers brought forth)~75%`
- `like:(Four score "and" seven years ago our fathers brought forth)~0.75`

All nine forms are equivalent and will return documents that have at least 8 (75% of 10 is 7.5 which is rounded up to 8) of the listed terms.

## Like Document Term Keyword Option

If the *like* term keyword option has a single term specified and that term has digits, or any punctuation, such as period, slash, colon, and so on, the term is assumed to be a *document id* and the most relevant terms will be extracted from that document and used as if they had been listed for the *like* term keyword option to boost relevancy for other documents containing those terms. This feature is sometimes referred to as "more like this" or "find similar" and is available in various commercial search engines.

A document ID is typically a web page URL, a file system path, a number, or some other special format, other than a term consisting of only letters, that uniquely identifies a given document. Most document IDs, including URLs, can be written directly, but the ID can be enclosed within quotes if it has any embedded spaces or to enhance readability. For example:

- Washington like:<http://cnn.com> -"New York"
- Washington like:"http://cnn.com" -"New York"

Both forms would select documents that contain "Washington" and do not contain "New York", with relevancy boosted by the most relevant terms contained in the web page at "http://cnn.com". This would find documents similar to the CNN web page, but requiring "Washington" and excluding "New York".

As a simple but reasonably detailed example, consider the following mini-documents in a Unix file system:

- /usr/home/jsmith/george.txt - George Washington
- /usr/home/jsmith/abe.txt - Abraham Lincoln
- /usr/home/jsmith/both.txt - George Washington and Abraham Lincoln

The following query would return george.txt and both.txt:

- like:/usr/home/jsmith/george.txt

That query is effectively the same as:

- like:(George Washington)

The following query would return all three documents:

- like:/usr/home/jsmith/both.txt

That query is effectively the same as:

- like:(George Washington Abraham Lincoln)

Note: Short words are ignored. The actual minimum word length is a configurable parameter.

The actual process of selecting the most relevant terms from the specified document is a bit more complex, but includes term frequency.

In addition, the terms are each given a calculated boost factor that corresponds to their calculated relevancy. The examples given here are simplified, but the actual queries include term weights based on frequency in the specified document.

The *like* option can be combined with the *minMatch* option to assure that only documents with some required percentage of terms are matched. For example, give these mini-documents:

- /usr/home/jsmith/alpha.txt - Alpha
- /usr/home/jsmith/beta.txt - Beta
- /usr/home/jsmith/gamma.txt - Gamma
- /usr/home/jsmith/alpha-beta.txt - Alpha Beta
- /usr/home/jsmith/beta-gamma.txt - Beta Gamma
- /usr/home/jsmith/alpha-gamma.txt - Alpha Gamma
- /usr/home/jsmith/all.txt - Alpha Beta Gamma

The following query would match all seven documents:

- like: "/usr/home/jsmith/all.txt"

The following query uses *minMatch* to select only those documents containing 66% or two-thirds of the relevant words extracted by the *like* option:

- like: "/usr/home/jsmith/all.txt"~2
- like: "/usr/home/jsmith/all.txt"~66%
- like: "/usr/home/jsmith/all.txt"~0.66

All three of those query forms are equivalent and will exclude the first three documents since they have too few of the optional terms.

The previous query is equivalent to this query:

- like: (Alpha Beta Gamma)~66%

## Query Parser Customization

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

The Lucid query parser offers a wide range of configuration settings, called *request parameters* that can be set in the Solr configuration XML file, `solrconfig.xml` (`solrconfig.xml` is specific to each collection. If using `collection1`, `solrconfig.xml` will be found in `$LWS_HOME/conf/solr/cores/collection1_0/conf`). After editing `solrconfig.xml`, LucidWorks Search should be restarted. On some Windows systems, it may be necessary to stop LucidWorks Search before editing any configuration file.

First, locate the `"/lucid"` *request handler*, which appears as follows:

```
<requestHandler class="solr.StandardRequestHandler" name="/lucid">
```

Next, locate the `"defaults"` entry, which appears as:

```
<lst name="defaults">
```

Not all of the configuration settings will be present in `solrconfig.xml`. If not present, the settings will default to internal default settings. In general, `solrconfig.xml` is used to *override* internal default settings.

Before adding an override setting you should scan the existing settings to see if there is already a setting that can be modified. If none is present, add a new entry for the setting as detailed below. The order of the settings does not matter, so new settings can simply be inserted after the `"defaults"` entry.

Each configuration setting entry has the following format:

```
<str name="sname">svalue</str>
```

where *sname* is the name of the setting, as detailed below, and *svalue* is the value of the setting. The setting value does not use quotes, even for string values.

A number of settings are Boolean on/off settings, where a value of *true* indicates that the setting is "on" or enabled, and *false* indicates that the setting is "off" or disabled.

## q.alt : Alternate Query

The `q.alt` setting specifies a default query to be used if the input query passed to the Lucid query parser is empty. By default, this setting is `*:*`, which selects all documents, which is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="q.alt">*:*</str>
```

To disable this behavior and simply select no documents to return no query results, use an entry as follows:

```
<str name="q.alt"></str>
```

## leadWild : Enable Leading Wildcards

The `leadWild` setting controls whether leading wildcards are permitted in queries. By default, this setting is "on" or enabled, which is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="leadWild">true</str>
```

To disable leading wildcards, turn this setting off, with an entry as follows:

```
<str name="leadWild">false</str>
```

## maxQuery : Query Limits

The Lucid query parser defaults to handling queries of up to 64K (65,536) characters in length. This should be sufficient to support even the most demanding of applications. But, should even this not be sufficient, configuration settings in `solrconfig.xml` may be added or modified to override these limits. In other cases, it may be desirable to dramatically decrease these limits to prevent rogue users from overloading the query/search servers in high-volume applications.

Here are the four configuration parameters that control maximum query length and their default values:

- `maxQuery` = 65,536 – Maximum length of the source query string (64K).
- `maxTerms` = 20,000 – Maximum number of terms in the source query string.
- `maxGenTerms` = 100,000 – Maximum number of Lucene Query terms that will be generated.
- `maxBooleanClauses` = 100,000 – Maximum number of Lucene Boolean Clauses that can be generated for a single BooleanQuery object. This includes original source terms, plus relevance-boosting phrases that are automatically generated.



Those default settings do not normally appear in `solrconfig.xml`, but if they did they would appear as follows:

```
<str name="maxQuery">65536</str>
<str name="maxTerms">20000</str>
<str name="maxGenTerms">100000</str>
<str name="maxBooleanClauses">100000</str>
```

If you wish to reduce the query length limit for maximum throughput of "casual" queries, a query length of 1,000 and 200 terms might be appropriate, which would require entries as follows:

```
<str name="maxQuery">1000</str>
<str name="maxTerms">200</str>
```

## nearSlop : Default Distance Limit for Proximity Operators

The `nearSlop` setting controls the default distance limit for the `NEAR`, `BEFORE`, and `AFTER` proximity operators. The internal default is 15, which is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="nearSlop">15</str>
```

The default distance can be changed to 10, for example, with an entry as follows:

```
<str name="nearSlop">10</str>
```

## notUp : Whether Upper Case is Required for the NOT Operator Keyword

The `NOT` keyword operator is a special case among the keyword operators since "not" is such a common word in natural language text and would be too easily confused with "not" as a keyword operator. For this reason, the `NOT` operator *must* be all upper case, unless the `notUp` request parameter is disabled to allow "not" as a lower case operator.

The `notUp` setting controls whether the `NOT` operator keyword must be all upper case. A setting of `true` means that the `NOT` keyword must be all upper case to be considered as an operator rather than as a simple text term. A setting of `false` means that the `NOT` operator keyword may be lower case or upper case, or even mixed case. The internal default is `true`, meaning that all upper case *is required* for the `NOT` operator keyword, which is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="notUp">true</str>
```

Lower case or mixed case for the `NOT` operator keyword can be enabled with an entry as follows:

```
<str name="notUp">false</str>
```

## opUp : Whether Upper Case is Required for Operator Keywords

The `opUp` setting controls whether operator keywords, such as `AND`, `OR`, and `NEAR`, must be upper case. A setting of `true` means the keywords must be all upper case to be considered as operators rather than simple text terms. A setting of `false` means that operator keywords may be lower case or upper case, or even mixed case. The internal default is `false`, meaning that upper case is not required, which is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="opUp">false</str>
```

Operator keywords can be required to be all upper case with an entry as follows:

```
<str name="opUp">true</str>
```

The `NOT` keyword operator is a special case since "not" is such a common word in natural language text and would be too easily confused with "not" as a keyword operator. For this reason, the `NOT` operator *must* be all upper case, unless the separate parameter, `notUp`, is disabled to allow "not" as a lower case operator.

## minMatch : Minimum Match of Optional Terms for Boolean Query

The `minMatch` configuration setting controls the minimum percentage of optional terms of a simple Boolean query that must match for a document to be selected. This configuration setting is used as the default unless the user explicitly uses the `minMatch` (or `atLeast`) keyword option or the tilde ('~') modifier on a simple Boolean query in the query string. A setting of 0 (the default) means that none of the optional terms is required for a document match. A value of 100 means that all optional terms must match. As a special case, any small percentage, such as 1, means that at least one optional term must match in any simple Boolean query. The default value of 0 is equivalent to placing this entry in the request "defaults" in `solrconfig.xml`:

```
<str name="minMatch">0</str>
```

To assure that at least one optional term matches in every simple Boolean query, use an entry as follows:

```
<str name="minMatch">1</str>
```

To require half (50%) of the optional terms to match in simple Boolean queries, use an entry as follows:

```
<str name="minMatch">50</str>
```

## 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](mailto: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