Search

Contents [show]

Introduction

The examples on this page apply to the "lucene" query language when using the search API.

Alfresco FTS is compatible with most, if not all of the examples here, except quoting of "@" and ":" etc is not required.

From Alfresco 3.4, Share exposes Alfresco FTS directly, and defines its own default macro that is usually used. However you can type in queries direct using the example here, remembering you are using Alfresco FTS.

Prior to Alfresco 3.4 the Share UI had its own custom query parser. The examples here do not apply.

The Search API

Searches are defined using the org.alfresco.service.cmr.search.SearchParameters object. They are executed using the public SearchService bean available from the RepositoryServices helper bean. A search returns aorg.alfresco.service.cmr.search.ResultSet which is itself made up oforg.alfresco.service.cmr.search.ResultSetRow's . Each row in a result set refers to a node in the repository. The rows returned in the result sets from the public SearchService bean are filtered to contain only the nodes to which the user executing the search has read access.

Search Parameters

A SearchParameters object allows the specification of:

- the search language
- the search string
- the definition of the fields ALL and TEXT
- the locale(s)
- bulk fetching of nodes from hibernate that are included in the results
- default AND or OR behavior
- the properties to be returned by the query
- query parameters
- sorting
- a limit to the size of the result set
- a limit to the number of permission evaluations carried out

- the generation of additional tokens for multilingual components of searches
- which store to search
- the transactional isolation level to use for the search

Currently, the supported search languages are:

- Lucene
- XPath

The specification of the search language to use is case insensitive.

The search string depends on the query language. Examples are given for each supported query language.

The data to search is specified by selecting which store to search. At present, this is restricted to a single store.

Query parameters define and substitute values into queries. The definition of query parameters depends on the query language. Examples are given for each supported query language.

The results from a search can be sorted using any property that can be recovered direct from the nodes that are found, but not properties of ancestors or dependents. There are special keys to sort by index order and relevance/score. By default search results are returned with the most relevant (such as highest score) first.

The maximum size of the result set can be set. The return results set will be no bigger than this size.

The maximum number of permission evaluations may also be set. Only this number of nodes will be considered for inclusion in the results set. If you set this to 10,000 and you do not have read permission for the first 10,000 things, you will see no results even though there may be a 10,001 node which you can see.

There is a time constraint applied to permission evaluations. This is set in the repository configuration, not on the QueryParameters.

The transactional isolation determines if the search sees information changed in the transaction or not. It will see all other data committed by other transactions. In database terms, the default behavior is READ_COMMITTED. If data changes made in the transaction are excluded from the search, data from all committed transactions will be found, excluding changes in the current transaction.

ALL and TEXT can be defined by the user.

The locale or locales are used to match the node locale â€” the locale of content and the locale of multi-lingual text properties. If there is more than one locale, query components are generated for each locale and tokenized accordingly.

There is no default filter for locale, primarily as locale is not present for most nodes. Non-localised properties (those which are not of type d:content or d:mltext) may exhibit unexpected behaviour when a node is not localised. The locale is found, in order of preference, from the node, from a thread local as set by the UI etc, and then from the Java default. If the thread local locale differs from the system default then tokenisation may change between initial index of a document and subsequent background FTS index (when there is no thread local locale). To avoid this in multi-lingual environments use a consistent tokenisation approach across all languages.

Example API Call

In this example, Lucene is used to find all nodes of the content type (using the first Lucene query example). It assumes you have access to the ServiceRegistry bean via Spring injection, or some other means.

```
...
        SearchParameters sp = new SearchParameters();
        sp.addStore(getStoreRef());
        sp.setLanguage(SearchService.LANGUAGE_LUCENE);
        sp.setQuery("TYPE:\"{http://www.alfresco.org/model/content/1.0}content\"");
        ResultSet results = null;
        try
        {
            results = serviceRegistry.getSearchService().query(sp);
            for(ResultSetRow row : results)
            {
                NodeRef currentNodeRef = row.getNodeRef();
                ...
            }
        }
        finally
        {
            if(results != null)
            {
                results.close();
            }
        }
...
```

It is important that the results set is closed after use, as some search implementations need this call to release resources. For example, the Lucene implementation releases IO resources.

try...finally pattern

The *try...finally* pattern in the previous example must be followed. If result sets are left unclosed they hold on to the underlying files in the Lucene index. Although merging can go ahead, the files no longer required after the merge will most likely not be deleted. The number of directories associated with the index will rise and no merged index directories will be deleted, so the disk space taken up by the index will also rise. Restarting will clean up any unused directories (in later versions of Alfresco which includes all the latest supported versions). If the code is not fixed to follow the required pattern the issue will recur.

At some point the Lucene result set will get bound to the transaction and closed automatically after the transaction commits. However it still makes sense to execute a query using the implicit transactions on the SearchService. In this case the result set returned may not be attached to a transaction and the *try...finally* pattern is required.

Config options for indexing

The configuration options for indexing are synchronous and asynchronous as defined on properties in the data dictionary. Some core information is always indexed atomically.

Disable in TX indexing and use index tracking. There will be latency issue similar to those found with the current clustering configuration.

Lucene

The Lucene query API is built on top the Lucene standard query parser. The basics of the query language can be found on the Lucene Web Site .

Data Dictionary Settings that affect indexing and searching

Currently, tokenization is set on the data dictionary type (such as d:text). This is picked up from a definition file that matches the locale of the server.

Understanding tokenization

If a property is both indexed and tokenized, as defined on the property, only the tokens generated from its string representation will go in the index. For example, using the standard configuration and running with an "en" locale, a d:text property will be tokenized using the AlfrescoStandardAnalyser.

"The fox's nose was wet and black, just like all the other foxes' noses. The fox liked to run and jump. It was not as brown or quick as the other foxes but it was very good at jumping. It jumped over all the dogs it could find, as quickly as it could, including the lazy, blackish dog."

AlfrescoStandardAnalyser

Text	Type	Start	End	Increment
fox	<APOSTROPHE>	4	9	1
nose	<ALPHANUM>	10	14	1
wet	<ALPHANUM>	19	22	1
black	<ALPHANUM>	27	32	1
just	<ALPHANUM>	34	38	1
like	<ALPHANUM>	39	43	1
all	<ALPHANUM>	44	47	1
other	<ALPHANUM>	52	57	1
foxes	<ALPHANUM>	58	63	1
noses	<ALPHANUM>	65	70	1
fox	<ALPHANUM>	76	79	1
liked	<ALPHANUM>	80	85	1
run	<ALPHANUM>	89	92	1
jump	<ALPHANUM>	97	101	1
brown	<ALPHANUM>	117	122	1
quick	<ALPHANUM>	126	131	1
other	<ALPHANUM>	139	144	1
foxes	<ALPHANUM>	145	150	1
very	<ALPHANUM>	162	166	1
good	<ALPHANUM>	167	171	1
jumping	<ALPHANUM>	175	182	1
jumped	<ALPHANUM>	187	193	1
over	<ALPHANUM>	194	198	1

all	<ALPHANUM>	199	202	1
dogs	<ALPHANUM>	207	211	1
could	<ALPHANUM>	215	220	1
find	<ALPHANUM>	221	225	1
quickly	<ALPHANUM>	230	237	1
could	<ALPHANUM>	244	249	1
including	<ALPHANUM>	251	260	1
lazy	<ALPHANUM>	265	269	1
blackish	<ALPHANUM>	271	279	1
dog	<ALPHANUM>	280	283	1

PorterSnowballAnalyser

Text	Type	Start	End	Increment
the	<ALPHANUM>	0	3	1
fox	<APOSTROPHE>	4	9	1
nose	<ALPHANUM>	10	14	1
wa	<ALPHANUM>	15	18	1
wet	<ALPHANUM>	19	22	1
and	<ALPHANUM>	23	26	1
black	<ALPHANUM>	27	32	1
just	<ALPHANUM>	34	38	1
like	<ALPHANUM>	39	43	1
all	<ALPHANUM>	44	47	1
the	<ALPHANUM>	48	51	1
other	<ALPHANUM>	52	57	1
fox	<ALPHANUM>	58	63	1
nose	<ALPHANUM>	65	70	1
the	<ALPHANUM>	72	75	1
fox	<ALPHANUM>	76	79	1
like	<ALPHANUM>	80	85	1
to	<ALPHANUM>	86	88	1
run	<ALPHANUM>	89	92	1
and	<ALPHANUM>	93	96	1
jump	<ALPHANUM>	97	101	1
it	<ALPHANUM>	103	105	1
wa	<ALPHANUM>	106	109	1
not	<ALPHANUM>	110	113	1
a	<ALPHANUM>	114	116	1
brown	<ALPHANUM>	117	122	1
or	<ALPHANUM>	123	125	1
quick	<ALPHANUM>	126	131	1
a	<ALPHANUM>	132	134	1
the	<ALPHANUM>	135	138	1
other	<ALPHANUM>	139	144	1

fox	<ALPHANUM>	145	150	1
but	<ALPHANUM>	151	154	1
it	<ALPHANUM>	155	157	1
wa	<ALPHANUM>	158	161	1
veri	<ALPHANUM>	162	166	1
good	<ALPHANUM>	167	171	1
at	<ALPHANUM>	172	174	1
jump	<ALPHANUM>	175	182	1
it	<ALPHANUM>	184	186	1
jump	<ALPHANUM>	187	193	1
over	<ALPHANUM>	194	198	1
all	<ALPHANUM>	199	202	1
the	<ALPHANUM>	203	206	1
dog	<ALPHANUM>	207	211	1
it	<ALPHANUM>	212	214	1
could	<ALPHANUM>	215	220	1
find	<ALPHANUM>	221	225	1
a	<ALPHANUM>	227	229	1
quickli	<ALPHANUM>	230	237	1
a	<ALPHANUM>	238	240	1
it	<ALPHANUM>	241	243	1
could	<ALPHANUM>	244	249	1
includ	<ALPHANUM>	251	260	1
the	<ALPHANUM>	261	264	1
lazi	<ALPHANUM>	265	269	1
blackish	<ALPHANUM>	271	279	1
dog	<ALPHANUM>	280	283	1

EnglishSnowballAnalyser

Text	Type	Start	End	Increment
the	<ALPHANUM>	0	3	1
fox	<APOSTROPHE>	4	9	1
nose	<ALPHANUM>	10	14	1
was	<ALPHANUM>	15	18	1
wet	<ALPHANUM>	19	22	1
and	<ALPHANUM>	23	26	1
black	<ALPHANUM>	27	32	1
just	<ALPHANUM>	34	38	1
like	<ALPHANUM>	39	43	1
all	<ALPHANUM>	44	47	1
the	<ALPHANUM>	48	51	1
other	<ALPHANUM>	52	57	1
fox	<ALPHANUM>	58	63	1
nose	<ALPHANUM>	65	70	1

the	<ALPHANUM>	72	75	1
fox	<ALPHANUM>	76	79	1
like	<ALPHANUM>	80	85	1
to	<ALPHANUM>	86	88	1
run	<ALPHANUM>	89	92	1
and	<ALPHANUM>	93	96	1
jump	<ALPHANUM>	97	101	1
it	<ALPHANUM>	103	105	1
was	<ALPHANUM>	106	109	1
not	<ALPHANUM>	110	113	1
as	<ALPHANUM>	114	116	1
brown	<ALPHANUM>	117	122	1
or	<ALPHANUM>	123	125	1
quick	<ALPHANUM>	126	131	1
as	<ALPHANUM>	132	134	1
the	<ALPHANUM>	135	138	1
other	<ALPHANUM>	139	144	1
fox	<ALPHANUM>	145	150	1
but	<ALPHANUM>	151	154	1
it	<ALPHANUM>	155	157	1
was	<ALPHANUM>	158	161	1
veri	<ALPHANUM>	162	166	1
good	<ALPHANUM>	167	171	1
at	<ALPHANUM>	172	174	1
jump	<ALPHANUM>	175	182	1
it	<ALPHANUM>	184	186	1
jump	<ALPHANUM>	187	193	1
over	<ALPHANUM>	194	198	1
all	<ALPHANUM>	199	202	1
the	<ALPHANUM>	203	206	1
dog	<ALPHANUM>	207	211	1
it	<ALPHANUM>	212	214	1
could	<ALPHANUM>	215	220	1
find	<ALPHANUM>	221	225	1
as	<ALPHANUM>	227	229	1
quick	<ALPHANUM>	230	237	1
as	<ALPHANUM>	238	240	1
it	<ALPHANUM>	241	243	1
could	<ALPHANUM>	244	249	1
includ	<ALPHANUM>	251	260	1
the	<ALPHANUM>	261	264	1
lazi	<ALPHANUM>	265	269	1
blackish	<ALPHANUM>	271	279	1
dog	<ALPHANUM>	280	283	1

ItalianSnowballAnalyser

Text	Type	Start	End	Increment
the	<ALPHANUM>	0	3	1
fox	<APOSTROPHE>	4	9	1
nos	<ALPHANUM>	10	14	1
was	<ALPHANUM>	15	18	1
wet	<ALPHANUM>	19	22	1
and	<ALPHANUM>	23	26	1
black	<ALPHANUM>	27	32	1
just	<ALPHANUM>	34	38	1
lik	<ALPHANUM>	39	43	1
all	<ALPHANUM>	44	47	1
the	<ALPHANUM>	48	51	1
other	<ALPHANUM>	52	57	1
foxes	<ALPHANUM>	58	63	1
noses	<ALPHANUM>	65	70	1
the	<ALPHANUM>	72	75	1
fox	<ALPHANUM>	76	79	1
liked	<ALPHANUM>	80	85	1
to	<ALPHANUM>	86	88	1
run	<ALPHANUM>	89	92	1
and	<ALPHANUM>	93	96	1
jump	<ALPHANUM>	97	101	1
it	<ALPHANUM>	103	105	1
was	<ALPHANUM>	106	109	1
not	<ALPHANUM>	110	113	1
as	<ALPHANUM>	114	116	1
brown	<ALPHANUM>	117	122	1
or	<ALPHANUM>	123	125	1
quick	<ALPHANUM>	126	131	1
as	<ALPHANUM>	132	134	1
the	<ALPHANUM>	135	138	1
other	<ALPHANUM>	139	144	1
foxes	<ALPHANUM>	145	150	1
but	<ALPHANUM>	151	154	1
it	<ALPHANUM>	155	157	1
was	<ALPHANUM>	158	161	1
very	<ALPHANUM>	162	166	1
good	<ALPHANUM>	167	171	1
at	<ALPHANUM>	172	174	1
jumping	<ALPHANUM>	175	182	1
it	<ALPHANUM>	184	186	1
jumped	<ALPHANUM>	187	193	1
over	<ALPHANUM>	194	198	1
all	<ALPHANUM>	199	202	1

the	<ALPHANUM>	203	206	1
dogs	<ALPHANUM>	207	211	1
it	<ALPHANUM>	212	214	1
could	<ALPHANUM>	215	220	1
find	<ALPHANUM>	221	225	1
as	<ALPHANUM>	227	229	1
quickly	<ALPHANUM>	230	237	1
as	<ALPHANUM>	238	240	1
it	<ALPHANUM>	241	243	1
could	<ALPHANUM>	244	249	1
including	<ALPHANUM>	251	260	1
the	<ALPHANUM>	261	264	1
lazy	<ALPHANUM>	265	269	1
blackish	<ALPHANUM>	271	279	1
dog	<ALPHANUM>	280	283	1

The VerbatimAnalyser just produces one token of type VERBATIM starting at position 0 and finished at 283.

Some words, such as "the", are excluded as tokens by some tokenizers. These are known as stop words. So if you try and search for the tokens "The" or "the", you will find nothing.

Tokenizers are language-specific, so are the stop words and the actual tokens generated, as they may be specific to the tokenizer. The tokens may not always be what you expect, particularly if the tokenizer uses stemming.

On the search side, all phrase queries will be tokenized. e.g. TEXT:"The quick Brown fox"

Term queries are also tokenized (post version 2.1). e.g. TEXT:Brown

Wildcard queries are lower case but not tokenized. e.g. TEXT:BRO*

(Wild cards (* and ?) are supported in phrases post version 2.1.) TEXT:"BRO*" will probably tokenize correctly and integrate with most stemming tokenizers, so long as BRO generates the stems required. The * should be ignored and removed by the analyser but understood by the QueryParser. If a particular analyser does not ignore * for stemming then phrase wildcards will not work.

Simple Queries

Summary of queryable fields

TEXT

By default all properties of type d:content. This can be user-defined.

ID

Id, where id is the full id, e.g:

```
ID:workspace\://SpacesStore/4ee34168-495a-46cd-bfd8-6f7590a4c0ce
```

ISROOT

T or F

ISNODE

T or F

TX

The transaction id

PARENT

Parent id - all child associations

PRIMARYPARENT

Parent Id - just the primary child association

QNAME

Association QName

CLASS

Class and all sub-classes

TYPE

Type and all sub-types

EXACTTYPE

Exact type

ASPECT

Aspect and all sub-aspects

EXACTASPECT

Exact aspect

ALL

Search all properties with appropriate tokenization

ISUNSET

Is the property unset

ISNULL

Is the property null

ISNOTNULL

Is the property

@prefix:name

Property

@{uri}name

Property

prefix:name

All properties of the given type

ur:name

All properties of the given type

PATH

Path (with no duplicates)

PATH_WITH_REPEATS

Path with duplicates (if there are multiple matching paths to a node, it will be repeated for all matching paths).

Fields you should not use and may be removed

ISCONTAINER
ISCATEGORY
LINKASPECT
PRIMARYASSOCTYPEQNAME
ASSOCTYPEQNAME
FTSSTATUS
ANCESTOR

The query parser

The query parser is a modification from the standard lucene query parser . The modifications are to support:

- wildcards at the start of term queries and additional fields, as described in this document.
- wildcards in phrases
- unbounded ranges
- datetime-based ranges

Note that queries with a single "NOT" entry or single TERM query preceded with "-" are not supported. For example,

```
-TYPE:"cm:object"
NOT TYPE:"cm:object"
```

Finding Nodes By Type

To find all nodes of type cm:content, including all subtypes of cm:content.

```
TYPE:"{http://www.alfresco.org/model/content/1.0}content"
TYPE:"cm:content"
TYPE:"content"
```

Note that local names containing invalid XML attribute characters should be encoded according to ISO 9075.

Finding nodes that have a particular aspect

To find all nodes with the cm:titled aspect, including all derived aspects from cm:titled.

```
ASPECT:"{http://www.alfresco.org/model/content/1.0}titled"
ASPECT:"cm:titled"
ASPECT:"titled"
```

Finding nodes by text property values

To find all nodes with the cm:name property containing the word banana:

```
@cm\:name:"banana"
```

Lucene requires the : to be escaped using the \ character. You must escape the escape character in Java like "@cm\\:name:\"banana\""

You can use the full {namespace}localName version of QName to identify the property but you will have a bit more escaping to do.

To find all nodes with the cm:name property containing words starting with "ban":

```
@cm\:name:ban*
```

To find all nodes with the cm:name property containing words ending with "ana":

```
@cm\:name:*ana
```

The standard Lucene query parser does not allow wild cards at the start for performance reasons. Use this with caution.

To find all nodes with the cm:name property containing words containing "anan":

```
@cm\:name:*anan*
```

To find all nodes with the cm:name property containing phrase "green banana":

```
@cm\:name:"green banana"
```

Finding nodes by integer or long property values

To find all nodes with the integer property test:int set to 12:

```
@test\:int:12
```

Note that leading zeros are ignored, so the following will also work.

```
@test\:int:00012
```

Long properties can be queried in a similar manner.

Finding nodes by float and double property values

To find all nodes with the property test:float equal to 3.4:

```
@test\:float:"3.2"
```

The search would be identical for double property values.

Finding nodes by boolean property values

To find all nodes with a boolean type property equal to TRUE:

```
@rma\:dispositionEventsEligible:true
```

The value "false" is the opposite of "true".

Finding nodes by content

Full text searches can be done in two ways. If the attribute cm:abstract is of type d:content, then its full text content will be indexed and searchable in exactly the same way as d:text attributes shown previously. You will be able to search against the text of the content, proving there is a translation to the mimetype text/plain.

So if cm:abstract contains the plain text "The quick brown fox jumped over the lazy dog", you could find the node holding the property using:

```
@cm\:abstract:"brown fox"
```

You can also use the TEXT field - this accumulates a broad full text search over all d:content type properties of the node. So to find nodes that have the word "lazy" in any content property:

```
TEXT:"lazy"
```

Content may not be indexed correctly for several reasons. Currently, these failures are recorded by inserting special tokens. These failures are not retried.

"nint"
: the content was not indexed as no appropriate transformation to text/plain was available.

"nitf"
: the content was not indexed as the transformation to text/plain failed with an exception.

"nicm"
: the content was not indexed as no content was found in the content store.

You can search for these tokens to find documents that had indexing problems. These tokens will also work in the UI search.

```
TEXT:"nint"
```

All uploaded content is tokenized according to the users locale. (It is not yet possible to specify locale on upload). At search time, the users locale is used for tokenization. Locale(s) can also be explicitly specified using the SearchParameters object.

A search is performed in the users locale except for d:mltext and d:content attributes which can be found in other locales as specified on the search parameters. Content will also be found in the locales specified on the search parameters if the tokens generated match those in other languages. The content part of the query only generates tokens in one language. However, those tokens are looked for in all the languages specified on the search parameters.

Finding nodes by content mimetype

All content type properties have an extended property to allow search by mimetype. This will be extended to include the size of the content in the future.

To find all cm:abstract content properties of mime type "text/plain":

```
@\{http\://www.alfresco.org/model/content/1.0\}content.mimetype:text/plain
```

Note: This extended attribute is not available for TEXT.mimetype as it may be made up of many different content types.

Finding nodes by date and time property values

Direct comparison of date/times can be problematic due to the the fractional part of the time. However, the index only contains the date part of the date-time at the moment. Dates are typically compared by range queries.

The query parser expects the date in ISO 8601 datetime format "yyyy-MM-dd'T'HH:mm:ss.sssZ". Truncated forms of this date are also supported.

You can also use the token TODAY to represent today's date and NOW to represent now.

So to find date and datetime properties:

```
@cm\:modified:"2006-07-20T00:00:00.000Z"
```

To search for a node with a date property cm:created equal to today:

```
@cm\:created:NOW
```

Finding nodes by QName

Each node is named within its parent. This name can be used as the basis of a search.

To find all the nodes with QName "user" in the name space with the standard prefix "usr":

```
QNAME:"usr:user"
```

This search type is related to path searches and only accepts name space prefixes and not full namespace entries. This is equivalent to the search PATH:"//usr:user"

The local name of a QName is ISO9075 encoded. To search for a QName of local name "space separated" and name space "example", use:

```
QNAME:"example:space_x0020_separated"
```

Searching Multilingual Text Fields

All multilingual fields are indexed and tokenized according to the locale. The tokens are prefixed in the index with locale information.

When searching, the locale(s) to use can be specified on the SearchParameters. If they are not specified, the locale defaults to the user's login locale (the locale selected when the user logged in).

The search is restricted to the specific strings in just those locales. By default, the locale "fr" will only match "fr" and not "fr_CA". How locales expand can be configured in the search parameters. "fr" can match "fr" only, or "fr" and all countries and all variants. "fr_CA_SomeVariant" can match only "fr_CA_SomeVarian" or "fr_CA_SomeVarieat", "fr_CA", and "fr".

In the default configuration, MLText is only identified by language - country and variants are ignored. So, searching in French will find all locales starting with fr.

If cm:mltitle were a ML string, it can be queried in Lucene using:

```
@cm\:mltitle:"banana"
```

The locales specified on the search parameters or the user's default locale govern what locales are matched.

The tokenization for each locale is picked up as defined by the data dictionary localization. By default, the locales are: default(en), cn, cs, da, de, el, en, es, fr, it, ja, ko, nl, no, pt_BR, pt, ru, and sv. Some locales have alternatives.

Path Queries

The path to a node is the trail of QNames of the child relationships to get to the node.

If the root node contains only one child called "one" in namespace "example", and this node has a child called "two" in namespace "example", the nodes in the repository can be identified by:

- "/"
- "/example:one" (a node specified by a child association "example:one" from the root node)
- "/example:one/example:two" (a node specified by a child association "example:one" from the root node, and then "example:two" from this node.)

This is very similar to attribute names and how they are specified in XPath. There is a special PATH field available to support queries against P.

Path queries support a subset of XPATH with the following axes:

- child
- descendant-or-self
- self

It supports the following node tests:

- name "name"
- namespace qualified name "prefix:name"
- "*" character

- namespace pattern "prefix:*"

You can not find all nodes regardless of namespace; "*:woof" is invalid.

It supports the standard abbreviations:

- . (self::node())
- // (descendant-or-self::node())

If omitted, the default aspect is "child::".

Predicates are not supported.

To find all nodes directly beneath the root node:

```
PATH:"/*"
```

To find all nodes directly beneath the root node in the "sys" namespace:

```
PATH:"/sys:*"
```

To find all node directly beneath the root node in the "sys" namespace and with local name "user":

```
PATH:"/sys:user"
```

To find all nodes directly below "/sys:user":

```
PATH:"/sys:user/*"
```

To find all nodes at any depth below "/sys:user":

```
PATH:"/sys:user//*"
```

To find all nodes at any depth below "/sys:user" including the node "/sys:user":

```
PATH:"/sys:user//."
```

To find the all nodes with QName "sys:user" anywhere in the repository:

```
PATH:"//sys:user"
```

To find all the children of all the all nodes with QName "sys:user" anywhere in the repository:

```
PATH:"//sys:user//*"
```

Important: If you have a large repository, increase the cache sizes to reflect common PATH queries. If not, your queries may be slower than expected.

Category Queries

Categories are treated as special PATHs to nodes.

There are not true child relationships between category type nodes and the things they categorize. However, these links can be searched using the special "member" QName. (If you try to follow these relationships via the node service this will not work.)

Categories themselves can be identified by a path starting with the QName of the aspect derived from "cm:classifiable" that defines them.

The following examples use the bootstrap categories. These are all categories in the "cm:generalclassifiable" classification.

To find all root categories in the classification:

```
PATH:"/cm:generalclassifiable/*"
```

If you know there is a "cm:Software Document Classification" category but you do not know at what level it exists, and you want the direct members: Note: If there are two categories with the same association QName, both will be found.

```
PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification/member"
```

If you know there is a "cm:Software Document Classification" category but you do not know at what level it exists, and you want all members:

```
PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification//member"
```

To find direct subclassifications of "cm:Software Document Classification" is more complex as you need to find any children that are not members of the category (such as things that are children but have not been categorized):

```
+PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification/*"
-PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification/member"
```

This finds any child that is not a "member" child.

To find all subclassifications of "cm:Software Document Classification" is more complex as you need to find any children that are not members of the category (such as things that are children but have not been categorized):

```
+PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification//*"
-PATH:"/cm:generalclassifiable//cm:Software_x0020_Document_x0020_Classification//member"
```

If you know "cm:Software Document Classification" is a root category you could miss the first //. For example, to find the direct members of the top level category "cm:Software Document Classification":

```
PATH:"/cm:generalclassifiable/cm:Software_x0020_Document_x0020_Classification/member"
```

Combined Queries

Any of the above queries can be combined using the Lucene standard methods.

The prefixes:

- - must not match (**Caution**: This is a restriction on the search results of the other terms)
- + must match
- (no prefix) may match - if there are only unprefixed clauses one must match

You may also use AND, OR and NOT.

To match two attributes:

```
+@test\:one:"mustmatch" +@test\:two:"mustalsomatch"
@test\:one:"mustmatch" AND @test\:two:"mustalsomatch"
```

To match one or other attribute:

```
@test\:one:"maymatch" @test\:two:"maymatch"
@test\:one:"maymatch" OR @test\:two:"maymatch"
```

To match one attribute and not another:

```
+@test\:one:"mustmatch" -@test\:two:"mustnotmatch"
@test\:one:"mustmatch" AND NOT @test\:two:"mustnotmatch"
```

However, you **cannot** just do a search on elements not matching a criteria:

```
-@test\:two:"mustnotmatch"
```

as it will not return any results (as it is a restriction on an empty set).

Instead, use a query similar to the following

```
+TYPE:"sys:base" -@test\:two:"mustnotmatch"
```

Any of the previous simple searches may be combined in these ways.

For example, to restrict a search by location in the hierarchy, and category, and full text search against title:

```
+PATH:"/cm:generalclassifiable/cm:Software_x0020_Document_x0020_Classification/member"
+@cm\:title:"banana"
+PATH:"/sys:user//*"
```

Parameterized Queries

Strings of the form ${namespaceprefix:variablename} are used to parameterize queries.

So to parameterize a full text search using the parameter "sys:text":

```
...
QueryParameterDefImpl paramDef = new QueryParameterDefImpl(QName.createQName("sys:text", namespacePrefixResolver) (DataTypeDefinition) null, true, "fox");
ResultsSet results = null;
try
{
   results = serviceRegistry.getSearchService().query(getStoreRef(), "lucene", "TEXT:\"${sys:text}\"", null,
        new QueryParameterDefinition[] { paramDef });
   ...
}
finally
{
   if(results != null)
   {
      results.close();
   }
}
...
```

This searches for nodes that contain "fox" in the TEXT. This does a straight replacement of pattern "${sys:text}" with "fox" in the query definition before executing the Lucene query.

Queries that sort

```
...
SearchParameters sp = new SearchParameters();
sp.addStore(getStoreRef());
sp.setLanguage(SearchService.LANGUAGE_LUCENE);
sp.setQuery("PATH:\"//.\"");
sp.addSort("ID", true);
ResultSet results = null;
try
{
    results = serviceRegistry.getSearchService().query(sp);
    for(ResultSetRow row : results)
    {
        NodeRef currentNodeRef = row.getNodeRef();
        ...
    }
}
finally
{
    if(results != null)
    {
        results.close();
    }
}
...
```

Lucene sorting is based on Java string ordering. Internally, we transform numeric and date types into a lexographical form that orders as expected.

Tokenising can give sort orders that may not be what you expect. Locale sensitive fields can only be ordered with respect to one locale. Nodes that are not relevant to the locale will be in the order they were added to the index (treated like null values in SQL)

Wild card queries

Wildcard queries using * and ? are support as terms and phrases. For tokenized fields the pattern match can not be exact as all the non token characters (whitespace, punctuation, etc) will have been lost and treated as equal.

Range Queries

Range queries follow the Lucene default query parser standard, with support for date, integer, long, float, and double types.

To search for integer values between 0 and 10 inclusive for the attribute "test:integer":

```
@test\:integer:[0 TO 10]
```

To search for integer values between 0 and 10 exclusive for the attribute "test:integer":

```
@test\:integer:{0 TO 10}
```

The constants 0 and 10 are tokenized according to the property type. So you could also use the previous search for long, float, and double types.

You can use the following for float and double ranges.

```
@test\:integer:[0.3 TO 10.5]
```

Date ranges

Dates are specified in "yyyy-MM-dd'T'HH:mm:ss.SSS" format only. A subset of ISO8601.

```
@test\:date:[2003\-12\-16T00:00:00 TO 2003\-12\-17T00:00:00]
@test\:date:[2003\-12\-16T00:00:00 TO MAX]
@cm\:created:[NOW TO MAX]
@cm\:created:[MIN TO NOW]
```

Date ranges accept truncated dates, but you can only truncate to the format to the day.

Unbounded range queries

If the upper or lower bound of a range does not tokenize to a sensible value for the property type, the query will be treated as unbounded.

```
@test\:integer:[3 TO MAX]
```

This is find any integer equal to or greater than 3.

By convention, MIN and MAX are good tokens for numeric and date types. For string types \u0000 and \uFFFF are good conventions.

Fields in the index

See above for the fields that should be used for queries. Using other internal fields is not supported, they are subject to change.

Note: When opening up Lucene Indexes with Luke tool, you may not always see all the indexed information. This is because Alfresco uses several indexes for each store to manage transactions and deletions. To understand overall Index Structure and how the different subindexes relate to each other see Index Version 2. The structure of indexes is as follows.

The Lucene index is split into two types of data:

- Properties and other key information held about nodes
- Additional information for nodes that contain other nodes

So a file would have one entry in the index for all of its properties and key information. A folder will have at least two entries in the index: one for all of its properties and key information, and one entry for each of the paths to the folder, a container entry. The container entries are used to support hierarchical queries.

Fields present on all entries

ID

The full noderef for the node to which the entry applies (ID:"workspace://SpacesStore/19f238df-7b5a-11dc-8388-991c49e2eac8").

Fields present on node entries

@{uri}localname

Fields of this form are created in the index for each property that is indexed and/or stored. The uri is the full name space uri of the property (not the prefix) and the local name the name of the property. For example, "@{http://www.alfresco.org/model/content/1.0}content ". When performing a search against the repository, well known prefixes defined by the models loaded by the data dictionary are available to identify attributes. So there are virtual fields of the form "@cm:content". The prefix form is converted into the previous full form to execute the query. The QueryParser will use the appropriate tokenisation method based on the type of the property as defined in the data dictionary.

Content has some additional information about the mimetype and size which can also be used in queries. These are of the form @cm:content.mimetype. (In future this will also support @cm:content.size for the size of the content in bytes and @cm:content.url for the internal content url.)

ASPECT

The aspects that have been applied to the node as the full QName string, e.g. {url}localname.

ASSOCTYPEQNAME

The type of the association from a parent.

FTSSTATUS

The status of the node index entry. This can be in three states:

Newâ€”the index contains a new entry but some properties have yet to be indexed in the background

Dirtyâ€”the node values have changed and some properties have yet to be indexed in the background

Cleanâ€”the index entry is up to date

Background indexing has to reindex all properties as Lucene does not support updates only add and delete.

ISNODE

Set to T. This identifies a node index entry.

ISROOT

Set to T only for the root node, and F for all other node entries.

LINKASPECT

For parent child relationships this contains ""; for catageory entries it contains the QName of the aspect that identifies the classification.

PARENT

Contains the parent ids for the node.

PRIMARYASSOCIATIONTYPEQNAME

The type of the primary association to the node.

PRIMARYPARENT

The primary parent of the node.

QNAME

The QNames of the node in each of the parents and categories that contain it.

TEXT

The accumulated full text search index.

TX

The transaction is in which the index was updated.

TYPE

The primary type of the node.

If a node has more than one parent or is categorized, there will be multiple values for ASSOCTYPEQNAME, LINKASPECT, PARENT, and QNAME. These entries are multi-valued, ordered with respect to each otherâ€”the first QNAME will be the name of the assocation in the first PARENT. The number of entries will be equal to the sum of the number of parents for the node "AND" and the number of categories into which the node has been placed.

Fields present on container entries to support hierarchical queries

ANCESTOR

This field stores the parent trail of node refs to the container.

PATH

A series of ordered QNames representing a path to the container.

ISCATEGORY

T if this node is a category.

ISCONTAINER

T as this is a container.

XPath

XPath queries support the contructs of Jaxen version 1.1, as it is implemented as document navigator against the NodeServce.

It has some JCR 170 specific extensions for functions, to support mutiple parents (which XML does not have), and to support sorting as required for JCR 170.

See Search_Documentation#XPath_search_using_the_node_service for more information about XPath.

Example Queries

Search all nodes starting from "company_home" having aspect "contractDocument" and with attribute "contractId" starting with 617, 858 or 984 :

```
+PATH:"/app:company_home//." AND +ASPECT:"{vic.model}contractDocument"
AND ( @vic\:contractId:"617*" OR @vic\:contractId:"858*" OR @vic\:contractId:"984*" )
```

Search all nodes starting from "company_home" having aspect "contractDocument" and attribute "referenceDate" equals to "2009-06-30T00:00:00" :

```
+PATH:"/app:company_home//." AND +ASPECT:"{vic.model}contractDocument"
AND +@vic\:referenceDate:"2009-06-30T00:00:00"
```

Example Queries That Sort

ISO 9075 encoding

XML attribute names can only start with and contain certain characters.

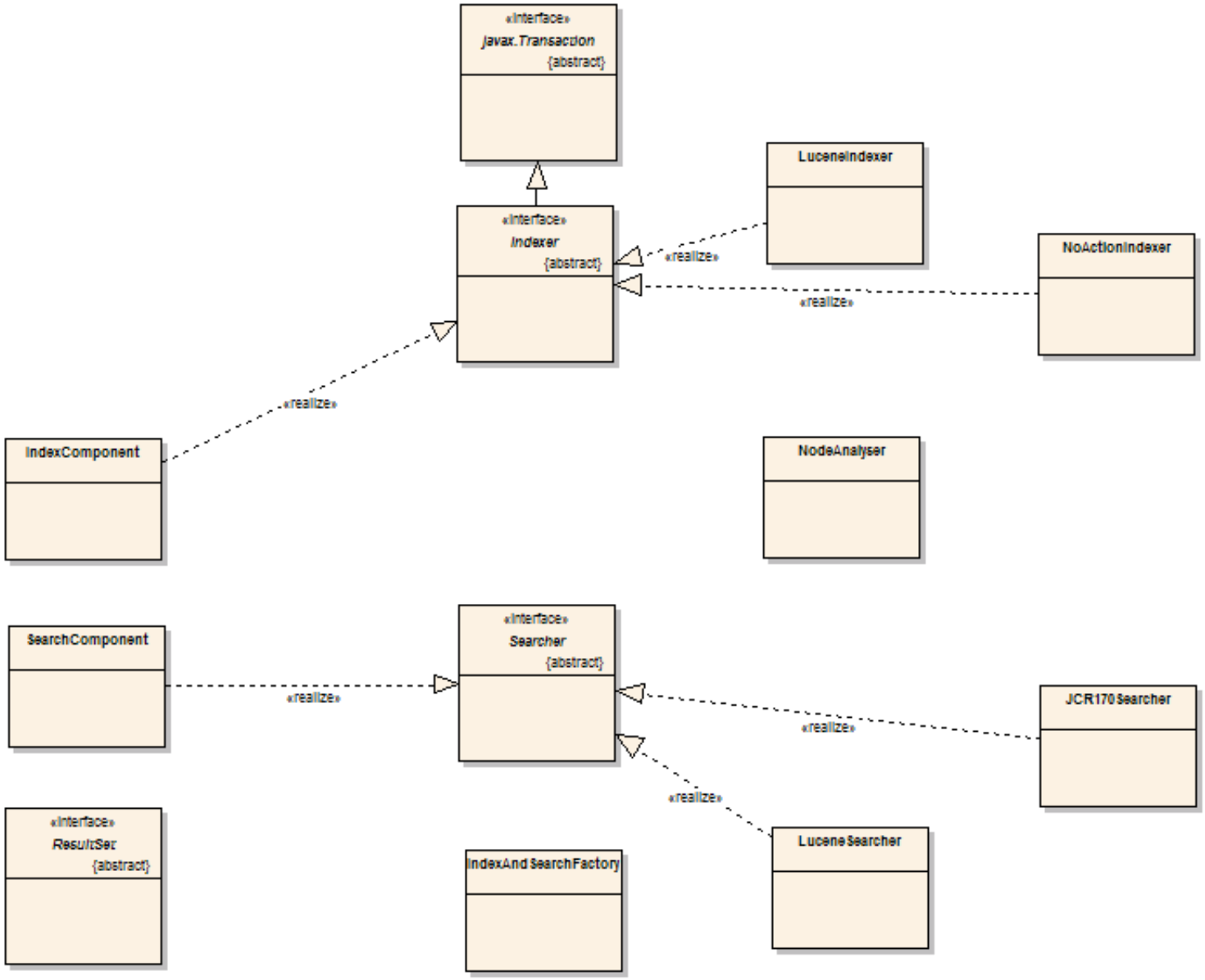
Invalid characters are encoded in the form "_xhhhh_" where the four digits represent the hex encoding of the character. For example, the space character, "a b", would get encoded to "a_x0020_b". If the initial string contains a pattern that matches "_xhhhh_", the first "_" is encoded. For example, "a_xabcd_b" is encoded as "a_x005f_abcd_b".

APIs

- Search Service API
- Query Web Service API
- Category API

There is no index service exposed as a public service.

Component Diagram



Implementation

- Index Version 1
- Index Version 2 - Available in versions 1.3, 1.3 E, and integrated in version 1.3.1 and 1.4

File Handles and Lucene

In UNIX environments the error "Too many open files" in the Alfresco log files is often associated with Lucene. The issue manifests itself on UNIX systems where often the system is configured by default to allow users a maximum of only 1024 open file handles, which is often not sufficient for the file structure used by the indexes.

Although Lucene 2.0 suffers less from this problem, errors may still occur on systems during periods of heavy use and therefore it is recommended that on production Alfresco systems the normal file handle limit is increased.

More information on how to verify that you are affected and recommended configuration changes can be found in Too many open files page.

Categories: Search | Lucene | Full Text Search | 3.3