Table of Contents

Author: Martin Bergljung

Introduction

This article is an update of the "Getting going with Solr in Alfresco 4" article that I wrote a couple of years ago. I thought it was time to update the article to cover new material added in Alfresco 5, Solr 4, Sharding etc. However, I have split all the search material in a basics article, this one, which will focus on the standard functionality available in the Community edition, like the first article did. And I will cover advanced stuff with Alfresco 5.1 and Solr in a special article called "*Searching with Alfresco 5.1 Enterprise*".
As in the first article I will assume that you have been using Alfresco for a while, and maybe you have even set up a proof of concept (PoC) project to play around with it to see how it all works, but you are not yet on top of Solr and wants to learn all about it. You might even have a client project waiting around the corner where you need to be up to speed on all the search functionality in Alfresco 5.1. Then it's time to dig into the Solr 4 search engine and have a look at how it is integrated with Alfresco and how it is configured. To use Solr with Alfresco is actually straight forward, just install Alfresco and by default it will use Solr for searching. Everything is done for you and Solr is quite transparent in this case. But this is just the way you would use it for testing and during PoC projects.
In a production environment you would want to separate Solr to run on its own server in its own Apache Tomcat application server. It is also useful to know about the different configuration files that Solr uses. So you can configure the cores, the schema, the caches, add synonym lists etc. This article will cover these things to a level where you can install Solr on a different server, configure it, tune it, and understand how it is integrated with Alfresco.

What is Solr?

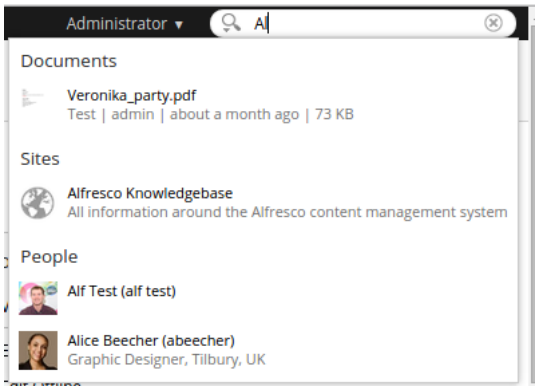
Apache Solr is an open source enterprise search server, or search engine if you like, that has been around long enough to be mature and power search on sites such as CNET and Netflix, see here for a more detailed list of who is using Solr. Current version is 5.5 but Alfresco 5.1 uses version 4.10. In the first article I wrote Alfresco 4 used Solr 1.4. So when you are reading up on Solr features for your Alfresco installation make sure you know the version of Solr that is used. Here is a short list of the new features in Solr 4:

- Advanced Search Capabilities (re-rank)
- Faceted Search & Filtering
- Auto-Suggest
- Spell Checking
- Statistics
- Geospatial
- Date Math
- Relevance
- Term-Highlighting
- Solr Cloud (Sharding/Replication)
- Caching (queries, filters, documents)

There is also more information about Solr 4 advantages in the Alfresco online documentation.
Note that the Apache Solr 4 server is supported only when running in an Apache Tomcat application server. Therefore, if you are running Alfresco within a different application server, then you must deploy Solr 4 to a separate Tomcat installation.
Under the hood Solr uses Apache Lucene as the indexing and search engine. Solr is written in Java and provides plug-in interfaces for building extensions to the search server. It can be run in an application server such as Apache Tomcat and you can talk to Solr via HTTP and XML, with it responding with XML or for example JSON.
Solr has the possibility to return a search result for a query (would not be much of a search engine otherwise) but it also has other features such as faceted searches and navigation as one can see on many e-commerce sites, results highlighting, "more like this" functionality for finding similar documents, query spell correction, query completion, and geospatial search for filtering and sorting by distance.

Why do we need a Search Engine such as Solr?

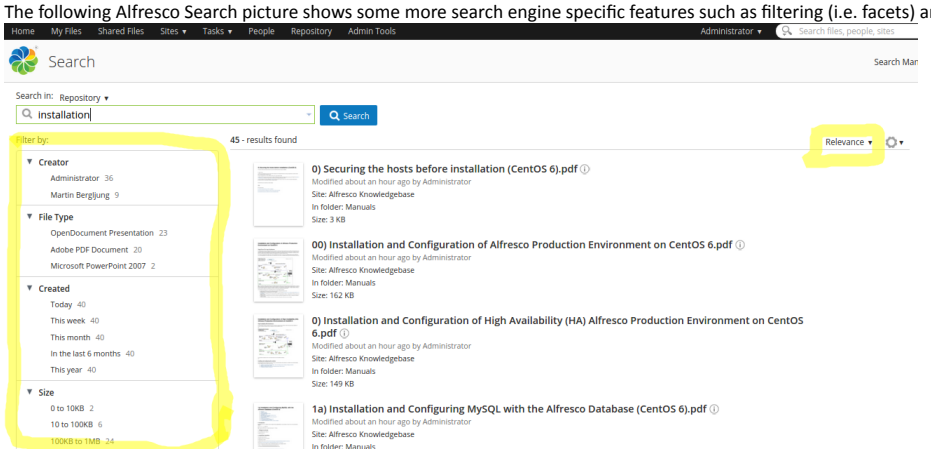
In this article I will assume that you don't know much about search at all. So you might be asking, why do I need a search engine at all, can we not just query the database for the information? Good question, take the following picture from Alfresco 5.1 search:



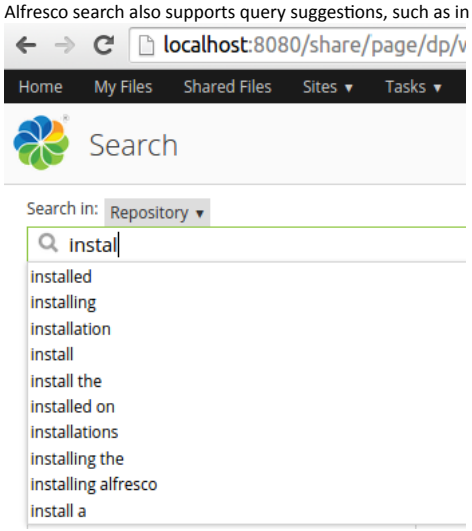
Here we are using the Instant Search feature in Alfresco, which provides an auto-complete like experience while you are typing your search word. It basically displays search results as you are typing. It searches documents, sites, and people instantly. The names of the site and the people could be looked up in the database. But the Free Text Search (FTS) in documents would not be possible to do via the database as the documents are stored in the filesystem.

A search engine such as Solr is optimized to handle textual data, which can come in multiple valid and invalid spellings, and names are often represented in many un-normalized forms. The index structure of a full text engines is more granular than for a database, allowing for rapid indexed access to specific words and phrases. Search engines also offer many more query operators, such as language stemming, thesaurus, fast wildcard matches, statistically based similarity and word densities, proximity, soundex and other "fuzzy" matching techniques. Further on, search engines provide relevancy weights to likely matches, allowing for much finer tuning of search results.

The following Alfresco Search picture shows some more search engine specific features such as filtering (i.e. facets) and sorting by relevance:

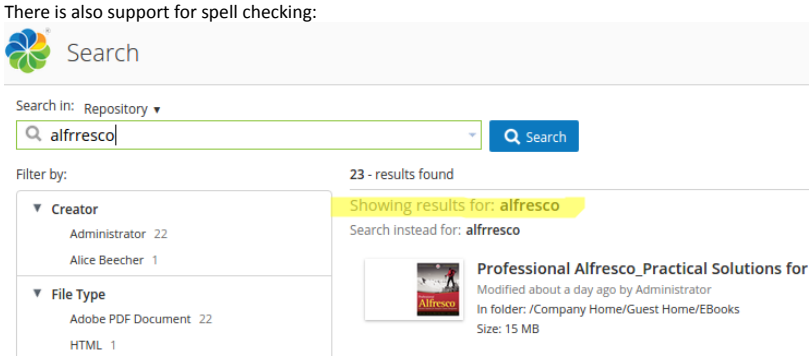


Alfresco search also supports query suggestions, such as in the following example:



While typing the query Alfresco will suggest a full query string to search on, up to 3 words.

There is also support for spell checking:



Features such as auto-complete, filtering, suggestion, spell checker, free text search (FTS) etc is only possible with a powerful search engine.

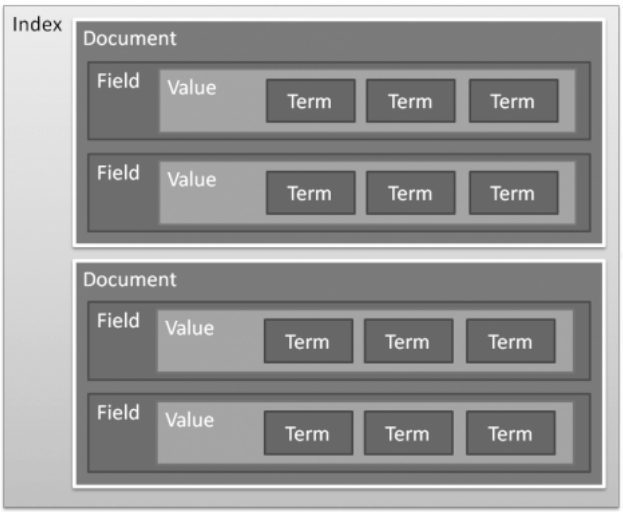
By now you are probably on-board with the advantages of using a search engine.
For a more in-depths comparison between search engines and databases read this article.

Solr and Lucene Concepts and Terminology

The search engine world uses a lot of new concepts and terminology that is good to get up to speed on before diving into the details.

Document, Fields, and Terms

This is the basic data format used by Apache Lucene for storage and retrieval. Documents are added and becomes searchable. Each document consists of one or more fields. Each field has a value, which in turn consists of one or more so called **terms**. A term represents a word from a text. This is the unit of search. It is composed of two elements, the text of the word, as **a string**, and the name of the field that the text occurred in.
The following picture illustrates:



To add a document to Lucene for indexing we would do something like this:

```
RAMDirectory directory = new RAMDirectory();
Analyzer analyzer = new StandardAnalyzer(Version.LUCENE_48);
IndexWriterConfig iwc = new IndexWriterConfig(Version.LUCENE_48, analyzer);
IndexWriter writer = new IndexWriter(directory, iwc);

Document doc = new Document();
doc.add(new Field("id", "001", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("someText", "The quick brown fox jumped over the lazy dog.",
            Field.Store.YES, Field.Index.ANALYZED));
writer.addDocument(doc);
writer.commit();
```

We will cover the details around fields, such as what it means to store a field, later on in this article. To search for documents in the Lucene index we would do something like this:

```
IndexReader reader = IndexReader.open(writer, true);
IndexSearcher searcher = new IndexSearcher(reader);
QueryParser parser = new QueryParser(Version.LUCENE_48, "someText", analyzer);
TopDocs td = searcher.search(parser.parse("fox"), 1000);
assertThat(td.totalHits, is(1));
```

When indexing and searching for documents via Solr, which is a web application wrapper around Lucene, an XML format is used to be able to transport the document over the wire with HTTP:

```
<add>
  <doc>
    <field name="id">001</field>
    <field name="someText">The quick brown fox jumped over the lazy dog.</field>
  </doc>
</add>
```

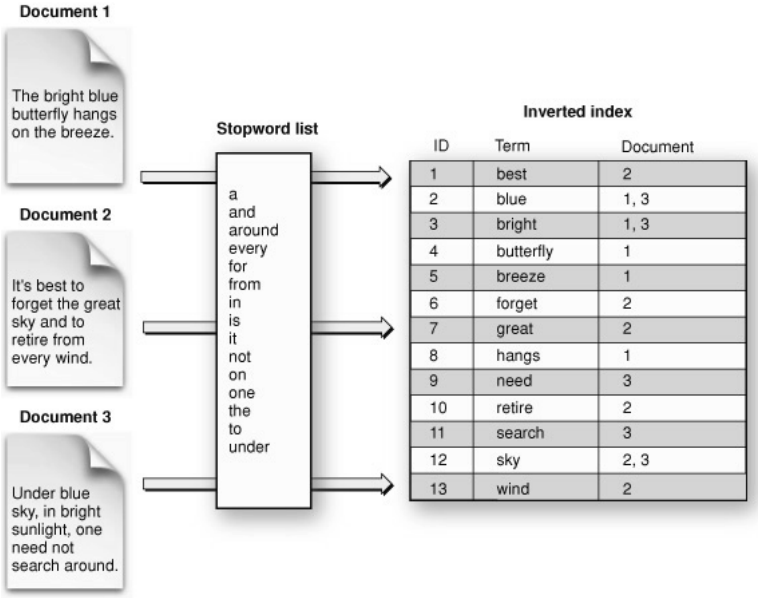
Having worked with Alfresco for a while it is easy to be confused as we also talk about storing documents in the Alfresco repository. But these documents are not the same kind of documents that are indexed by a search engine. Documents in Alfresco are more like files.

Document Numbers

Internally, Lucene refers to documents by an integer **document number**. The first document added to an index is numbered zero, and each subsequent document added gets a number one greater than the previous.

Inverted Index and Stop Words

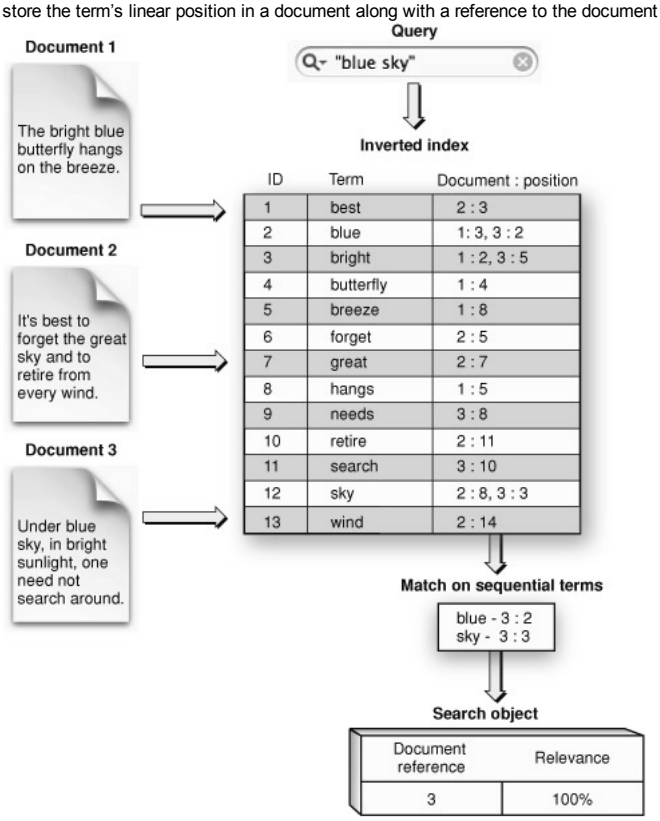
When we do a text search with Solr, and indirectly Lucene, it is able to respond very quickly because it does not actually search through the complete text of all documents in the system. Instead it uses an index to search. To understand this think about how an index in a book works, you look up a word in the book index, and it will tell you on what pages you will find this word. This kind of index is called an **inverted index** and this is what is used in Solr, or more specifically in Lucene.
The Lucene index consists of words, also called **terms**, that points to the documents that they can be found in:



Here we can see that certain words/terms are filtered out before we index the text. These are referred to as **stop words**. They are words that you are unlikely to want to search on, such as “a”, “for”, “on” etc. A search on one of these words would just match too many documents to be able to get a relevant/useful response back.

Inverted Index supporting Phrase and Proximity Searches

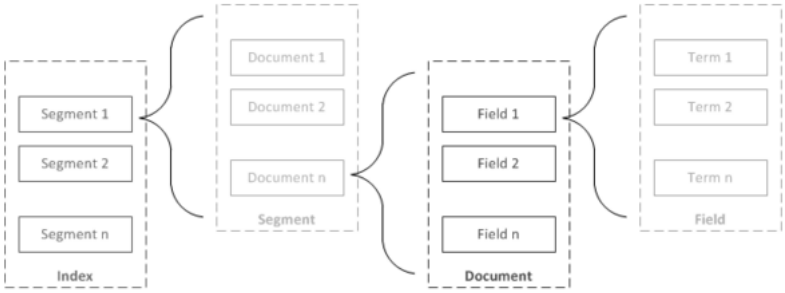
You might be familiar with phrase searches (e.g. "big yellow banana") and proximity searches (e.g. “big * apple”) in Alfresco. An index that supports these types of searches will also store the term's linear position in a document along with a reference to the document the term appears in:



Searching for a phrase such as "blue sky" means searching for a series of terms that appear in consecutive order. Similarly, searching for proximity of words, such as “blue * sky”, means to search for a pair of terms whose linear distance is small.

Segment

Lucene indexes may be composed of multiple sub-indexes, or **segments**. Each segment is a fully independent index, which could be searched separately. Indexes evolve when new segments are created for newly added documents and when merging existing segments. Searches may involve multiple segments and/or multiple indexes, each index is potentially composed of a set of segments.
The following figure shows the basic structure:



Core

When talking about Solr you will quite early on hear people mentioning something called Solr cores. It is important to know what a Solr core is before moving on. Previously when working with Alfresco and its embedded Lucene index engine you were talking about “the” index. With Lucene there is one index and that’s it. Not so with Solr. A Solr core holds one Lucene index and the supporting Solr configuration for that index; sometimes the word "core" is used synonymously with "index".
The Solr server can manage multiple cores, meaning it can manage multiple Lucene indexes. Nearly all interactions with Solr are targeted at a specific core. Using multiple cores has some advantages in a production environment:

- You can rebuild an index (i.e. core) offline with minimal impact on performance, and the offline index could then also be optimized for updating content
- You can test configuration changes to an index without affecting the live environment by copying the live index to a test index
- You can rename cores at runtime and keep multiple versions of the same core, deciding which one should be used by the end users

There are also other advantages such as the possibility to index other content sources, in addition to the Alfresco repository, in a new core and have it searchable via the Alfresco Solr installation.

Doc Values

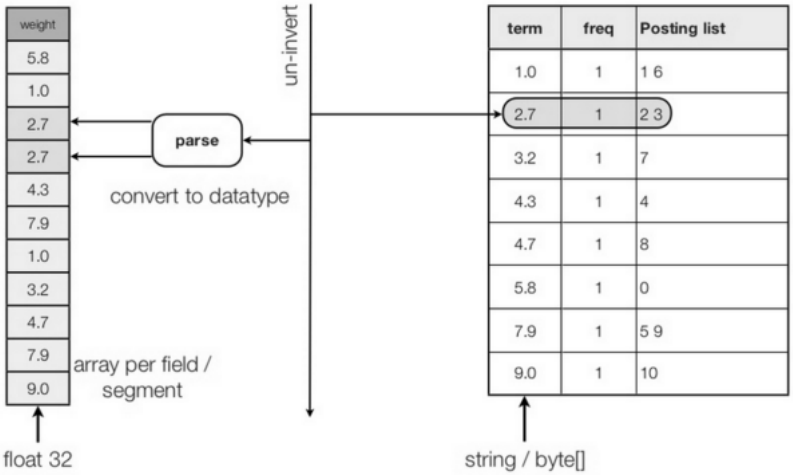
The standard way that Lucene builds the index is with an inverted index. This style builds a list of terms found in all the documents in the index and next to each term is a list of documents that the term appears in (as well as how many times the term appears in that document). This makes search very fast - since users search by terms, having a ready list of term-to-document values makes the query process faster.
For other features that we now commonly associate with search, such as sorting, faceting, and highlighting, this approach is not very efficient. The faceting engine, for example, must look up each term that appears in each document that will make up the result set and pull the document IDs in order to build the facet list. In Lucene, this is maintained in the Field Cache in memory (internal Lucene cache), and can be slow to load (depending on the number of documents, terms, etc.).
So what we have with an inverted index is this:

- **Term1** -> docId1, docId2
- **Term2** -> docId2
- **Term3** -> docId1,docId2

What we need is something like this:

- **docId1** -> Term1, Term3
- **docId2** -> Term1, Term2, Term3

For simplicity we can think of the Field Cache as an array indexed by Lucene’s internal document number (ID). Here is how this is done by “un-inverting” the index and creating the Field Cache:



So here Lucene is un-inverting a field called `weight` into the Field Cache. Note that there has to be a type conversion between String and float (until Lucene 4). Loading the field cache takes time and requires a lot of memory, and it also builds an unnecessary term dictionary.
In Lucene 4.0, a new approach was introduced called **DocValue** fields, which are column-oriented fields with a document-to-value mapping built at index time. This approach promises to relieve some of the memory requirements of the Field Cache and make lookups for faceting, sorting, and grouping much faster.
Here is an example of how to add a DocValue field called `pageRank` at index time:

```
Document doc = new Document();
doc.add(new Field("id", "001", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("someText", "The quick brown fox jumped over the lazy dog.",
    Field.Store.YES, Field.Index.ANALYZED));

float pageRank = 10.3f;
DocValuesField dvField = new DocValuesField("pageRank");
dvField.setFloat(pageRank);
doc.add(dvField);

writer.addDocument(doc);
writer.commit();
```

To get the page ranking for a document is now quite easy:

```
DocValues docValues = reader.docValues("pageRank");
```

```
int docId = 0;
double pageRankForDoc = docValues.getSource().getFloat(docId);
```

This reads directly from disk and caches the array with values in memory.
Sometimes the DocValue field should also be indexed, stored, or needs term vectors:

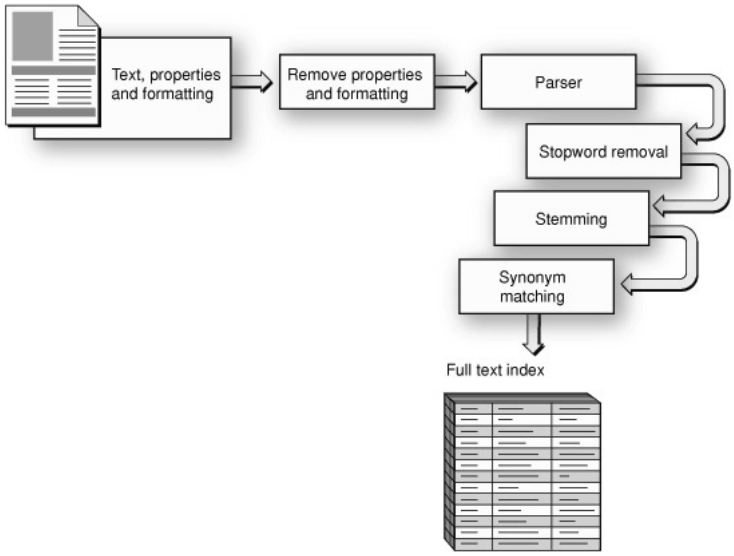
```
Document doc = new Document();
doc.add(new Field("id", "001", Field.Store.YES, Field.Index.NOT_ANALYZED));
doc.add(new Field("someText", "The quick brown fox jumped over the lazy dog.",
    Field.Store.YES, Field.Index.ANALYZED));

String theTitleText = "The quick brown fox";
DocValuesField dvField = new DocValuesField("title");
dvField.setBytes(new BytesRef(theTitleText), Type.BYTES_VAR_DEREF);
Field titleField = new Field("title", theTitleText, Field.Store.YES, Field.Index.ANALYZED);
titleField.setDocValues(dvField);
doc.add(titleField);

writer.addDocument(doc);
writer.commit();
```

The more filesystem cache space that you have available, the better DocValues will perform. If the files holding the DocValues are resident in the filesystem cache, then accessing the files is almost equivalent to reading from RAM. And the filesystem cache is managed by the kernel instead of the JVM.
Here is a summary of the major advantages of using Doc values instead of the Field Cache:

- They live on disk instead of in heap memory. This allows you to work with quantities of field data that would normally be too large to fit into memory. This improves the speed of garbage collection and consequently stability.
- Doc values are built at index time, not at search time. While in-memory field data has to be built on the fly at search time by “uninverting” the inverted index, doc values are prebuilt and much faster to initialize.

Text (Term) Extraction

The document text is not added directly to the index, it usually goes through a series of transformations before it is added to the index. This is referred to as the **analysis phase**.
Transformations could be things like converting to lowercase, doing stemming, stop word filtering etc. The end result of the analysis are a series of **terms** which are added to the index.
Terms, not the original text, are what are searched when you perform a search query.
The following figure illustrates:



The various components involved in text analysis and extraction go by various names, such as analyzers, tokenizers, filters etc. They are all conceptually the same, they take in text and output text, sometimes filtering, sometimes adding new terms, and sometimes modifying terms. The difference is in the specific flavor of input and output for them: either character based or token based.
Also, term , token , and word are often used interchangeably.
An analysis phase is also used when we search, typically matching the index analysis phase.

Tokenization

A tokenizer is a component in the analysis phase that takes text in the form of a character stream and splits it into so called **tokens**. The tokenizer will skip insignificant bits like whitespace and joining punctuation.

Terms Dictionary

The Term Dictionary is a file that stores how to navigate the various other files for each term. Terms are stored in alphabetical order for fast lookup. Other bookkeeping (skip lists, index intervals) is also tracked with the term dictionary.

Term Frequency

The number of times a term occurs in a document is called its term frequency. Term frequency information is used to determine which document is most relevant to a given search query.

Term Vector (Positions)

Term vectors are terms, along with their frequency count and positions. The term vector stores information generated by the indexing process. The index is a map from terms to documents and the term vector is a map from terms to position, offset, and frequency information in the document that the term belongs to.
With the term vector information new types of searches can be supported, such as result highlighting. To generate this information we need to know where the search terms occurs in the document, so we can select the piece of texts around the search term and optionally highlight the term. The term vector contains all the necessary information to let us do this.

Another interesting thing we can do with term vector information is to find similar documents of a given document, this is often referred to as the “More Like This” feature.

Scoring

Lucene scoring is the process of determining how relevant a given document is to a user's search query and present it accordingly in the search result, if no sorting has been applied. The score of each document is represented by a positive floating-point number. The higher the score number, the more relevant the document.
It is important to note that Lucene scoring works on fields and then combines the results to return documents. This is important because two documents with the exact same content, but one having the content in two fields and the other in one field will return different scores for the same query due to length normalization.
The basic scoring factors:

- **Term Frequency** - the more times a search term appears in a document, the higher the score
- **Inverse Document Frequency** - matches on rarer terms count more than matches on common terms
- **Coordination factor** - if there are multiple terms in a query, the more terms that match, the higher the score
- **Length Norm** - matches on a smaller field score higher than matches on a larger field

Score Boosting

You might have encountered boosting when doing searches on websites on the Internet. A good example is Google in itself where you are shown some search results boosted to the top (or a place which catches your attention) as they would be from a sponsored source. In essence Google has boosted the sponsored search result to the top to bring it to prominence. Lucene allows influencing the search result by “boosting” at different times.

Relevance

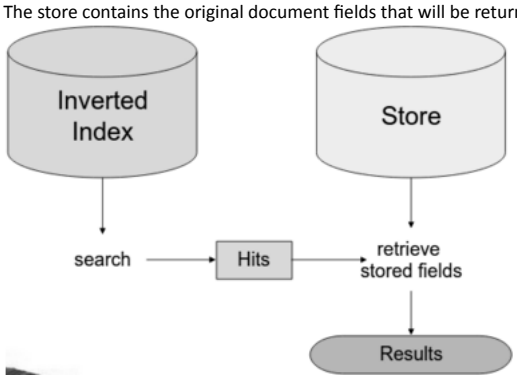
Relevance is the degree to which a query response satisfies a user who is searching for information. A query clause generates a score for each document. How that score is calculated depends on the type of query clause. Different query clauses are used for different purposes: a fuzzy query might determine the score by calculating how similar the spelling of the found word is to the original search term; a terms query would incorporate the percentage of terms that were found. However, what we usually mean by relevance is the algorithm that we use to calculate how similar the contents of a full-text field are to a full-text query string.
This is linked to term frequency and scoring. For example, when searching for term ‘x’ and it appears 25 times in a field in document D1 and 1 time in the same field in document D2, then the D1 document is more relevant and gets higher ranking when presenting the search result.

Norms

A norm is part of the calculation of a score. The norm could actually be calculated however you like. The main thing that sets the norm apart, is it's calculated at index-time. Generally, other factors influencing score are calculated at query time, based on how well the document matches the query. The norm saves on query performance by being stored along with the document. When in memory, norms occupy one byte per document for each field with norms on, even if only one document has norms on for that field. The standard implementation can be found, and well described, in Lucene's TFIDFSimilarity.
Norms can be controlled via the `omitNorms` property for the field definitions in the schema. If `true`, then it omits the norms associated with this field (this disables length normalization and index-time boosting for the field, and saves some memory). Only full-text fields or fields that need an index-time boost need norms.

Store

The store contains the original document fields that will be returned in search results. The following figure illustrates:



The Solr schema can be used to set up a field to be stored or not. The more fields that are stored, the more disk space will be needed.

Faceting

Facets are used to organize information in multiple ways, often based on structured information. Primary use-case is to narrow down a search result to easier find what you are looking for. More specifically, in Solr faceting is the arrangement of search results into categories based on indexed terms. Searchers are presented with the indexed terms, along with numerical counts of how many matching documents were found for each term.

Suggesting

The suggest component in Solr provides users with automatic suggestions for query terms. You can use this to implement a powerful auto-suggest feature in your search application. When typing a query in Alfresco Share search it will suggest a full query string to search on, up to 3 words.

Highlighting

Highlighting in Solr allows fragments of documents that match the user's query to be included with the query response. The fragments are included in a special section of the response (the highlighting section), and the client uses the formatting clues also included to determine how to present the snippets to users.

Ranking

The higher rank a document have in the search result the closer to the top of the result display it will have. The highest ranked document will be at the top of the search result. Ranking is determined by the scoring of a document and is referred to as relevance ranking. When sorting criteria is specified this overrides the ranking.

Re-Ranking

Query Re-Ranking allows you to run a simple query (A) for matching documents and then re-rank the top N documents using the scores from a more complex query (B).
For example, let's say you are searching for "The AND big AND red AND banana". This might give loads of results back. To narrow down and present the most accurate and useful search result we search again in the top 1000 results for "The big red banana" (doing a re-rank).

Phrasing and Auto-Phrasing

A phrase is more than one word put together in a fixed order. Phrase search is a type of search that allows users to search for documents containing an exact sentence or phrase rather than containing a set of keywords in random order. Phrase searches are usually done by using quotation marks (") around a specific phrase to indicate that you want to search for instances of that search query.
Note though that searching for phrases is really slow as you have to use term position.
Auto-Phrasing is used when the user is leaving out the quotation marks (") around the search query. Then a re-rank is done with a quoted phrase.

Shingle

Large content repositories usually contain many documents that are similar but not identical. To find these related documents, you can apply the process of shingling. Shingling extracts the textual essence from each document and applies that pattern to a data set to find similar documents. This can help you speedily reduce the number of files to review or find the changes made between very similar files.
A shingle is effectively a word-nGram. Given a stream of tokens, a shingle filter will create new tokens by concatenating adjacent terms. For example, given the phrase "the dog smelled like a skunk", a shingle filter might produce:

- "the dog"
- "the dog smelled"
- "dog smelled like"
- "smelled like a"
- "like a skunk"
- "a skunk"

The shingle filter allows you to adjust min_shingle_size and max_shingle_size, so you can create new shingle tokens of any size.
Shingles effectively give you the ability to pre-bake phrase matching. By building phrases into the index, you can avoid creating phrases at query time and save some processing time/speed.
The downside is that you have larger indices and potentially more memory usage if you try to facet/sort on the shingled field.
The suggester functionality in Alfresco is backed by the shingle filter.

Schema

An index is represented by one or more **documents**. A document consists of one or more **fields**, which together represents a set of data describing something like a patient record, cooking recipe, person, marketing document, drawing, engineering document etc. So a document in the Solr/Lucene world is different from what we think of as a document, or file, in the Alfresco world.
Fields in a Lucene index are created as you go; adding a document to an empty index with a numeric field named "price," for example, makes the field instantly searchable, without prior configuration.
When indexing a lot of data it is however often a good idea to predefine a set of fields and their characteristics to ensure consistent indexing. To allow this, Solr adds a **data schema** on top of Lucene, where fields, data types, and data analysis chains can be precisely defined.
Before you can add any documents to Solr you need to set up a schema, which tells Solr what fields to expect in the document, which field that is the primary key field (i.e. unique between all documents), what fields that are required, how to search and index each field and so on.
A field definition in the Solr schema looks like this (<alfrescoinstalldir>/solr4/workspace-SpacesStore/conf/**schema.xml**) :

```
<schema name="Alfresco V2.0" version="1.5">
 ...
  <fields>
        <field name="id" type="identifier"  indexed="true"  omitNorms="true"  stored="true"   multiValued="false"  required="true"  docValues="true"/>
```

This field is from the Solr schema used by Alfresco and represents the unique field in documents posted to Solr from Alfresco. The different attributes in the field specification have the following meaning:

- **name**: name of the field
- **type**: field data type (determines how the data is analyzed when it is added to the index and how the data is processed searching the index)
- **indexed**: should this field be analyzed and added to the inverted index?
- **stored**: should the original value of this field be stored and returned in search results?
- **multiValued**: can this field have multiple values?
- **required**: is this field required to have a value?
- **docValues**: are a way of recording field values internally at index time that is more efficient for some purposes, such as sorting and faceting, than traditional indexing
- **omitNorms**: A norm is part of the calculation of a score and it is **calculated at index-time**. For example, if **omitNorms** is set to **false**, and one document is named "Karate Masters of Tomorrow", and another document is named "Karate Kid" and I search for "Karate", then the "Karate Kid" document will score higher in the result list as it has only two words in the name.

So, shouldn't the **indexed** attribute always be **true**, what's the purpose of a field if you don't index it and make it searchable? There are situations when we might want to return a field value in a search result even though it is not used for searching.
To summarize the **indexed** and **stored** attributes:

- An **indexed** field is a field which is searchable and sortable. Indexed fields are **not** returned in the search results.
- A **stored** field's value is returned in search results.
- If a field is both **indexed** and **stored**, then the field is searchable, sortable, and returned in search results.

These attributes can also be found in the Alfresco content model when you configure indexing behavior, as in the following example:

```
<property name="acme:securityClassification">
  <type>d:text</type>
  <index enabled="true">
    <atomic>true</atomic>
    <stored>false</stored>
    <tokenised>false</tokenised>
    <facetable>true</facetable>
  </index>
```

When we talk about Solr it is also worth mentioning that the atomic attribute in the above index configuration is not used. Setting atomic to true means that you want the indexing to happen in-transaction (i.e. indexing happens in the same transaction as for example a document upload), which is not possible with Solr, only with Alfresco installations that uses Lucene by itself. Solr is tracking the Alfresco database transactions and will eventually catch up and be consistent.
Setting stored to true does not have any effect now either with the Alfresco 5.1 and Solr 4 integration as the complete document is always cached in the new Solr caching content store.
We tell Solr about the kind of data a field contains by specifying its **field type**. The **id** field is of the type **identifier**, which looks like this:

```
<types>
        <fieldType name="identifier" class="solr.StrField" sortMissingLast="true" positionIncrementGap="100" />
```

This field type is implemented in a Java class pointed to by the **class** attribute, which is set to **solr.StrField** in this case (it is not necessary to include the full package name, it is resolved to `org.apache.solr.schema`).
So the data for the **id** field is going to be a string that is UTF-8 encoded or in Unicode. So, can you do wildcard search on this kind of string field? No you cannot. A **solr.StrField** cannot have any tokenization, analysis, or filters applied, and will only give results for exact matches. If we need a string field with analysis or filters applied, then we can implement this using a **solr.TextField**, which is what is used for other string fields where it should be possible to do wildcard searches.
What this all means is that you cannot upload and index a document where there are unknown fields (i.e. they are not in the schema). Next question you have now is probably, how about running schemaless and figure out the schema on the fly, can Solr do that? Yes, it can actually from version 4.3. But this is not used in the Alfresco -> Solr integration.
What about custom content models that we deploy to Alfresco, do we need to update the schema every time we deploy a new content model? No you don't, there is another feature in Solr called **dynamic fields** that is used to handle this. Dynamic fields allow Solr to index fields that we did not explicitly define in the schema. This is useful if we have forgotten to define one or more fields, or as in the case of Alfresco we don't know all the possible custom fields (i.e. properties) that will be defined in future custom content models.
A dynamic field is just like a regular field except it has a name with a wildcard in it. When you are indexing documents, a field that does not match any explicitly defined fields can be matched with a dynamic field. For example, the schema in the Alfresco Solr integration has many dynamic field definitions similar to this one:

```
<dynamicField name="text@s___@*"        type="identifier"       indexed="true" omitNorms="true"   stored="false"  multiValued="false" sortMissingLast="true" />
```

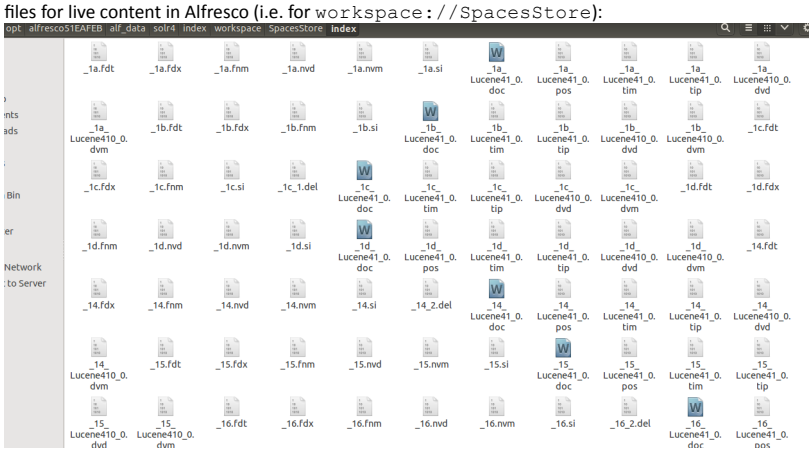
So if we post a field for indexing and it has the following name:

```
text@s___@{http://www.alfresco.org/model/content/1.0}name
```

Then it will be matched up with the above dynamic field definition. We will go into these in more detail later on.

Lucene File Formats

When we look at the files on disk (<alfrescoinstalldir>/alf_data/solr4/index/...) that make up the Lucene index we can see that there are loads of them. Here is an example of the index files for live content in Alfresco (i.e. for `workspace://SpacesStore`):



All files belonging to a segment have the same name with varying extensions. Here is a list of most of the extensions that we can see in an Alfresco 5.1 installation that uses Lucene 4:

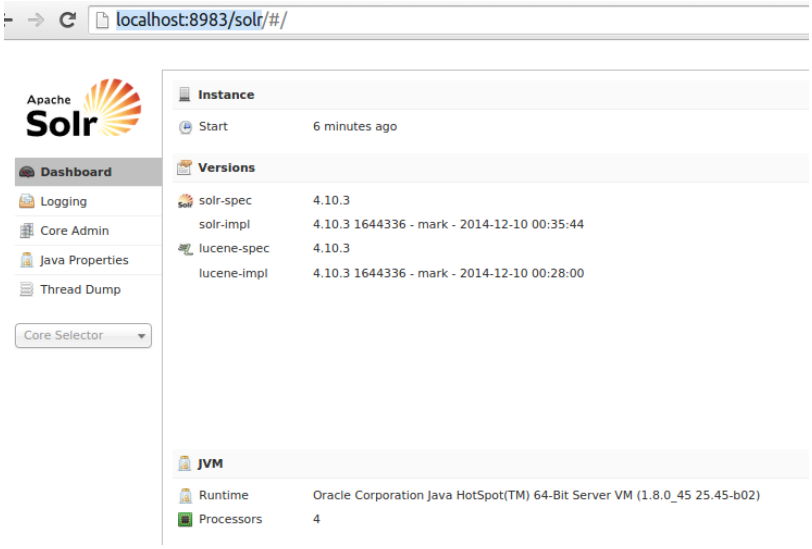
Name	Extension	Description
Segment Info	.si	Stores metadata about a segment
Fields	.fnm	Stores information about the fields
Field Index	.fdx	Contains pointers to field data
Field Data	.fdt	The stored fields for documents
Term Dictionary	.tim	The term dictionary, stores term info
Term Index	.tip	The index into the Term Dictionary
Term Positions	.pos	Stores position information about where a term occurs in the index
Frequencies	.doc	Contains the list of docs which contain each term along with frequency
Norms	.nvd,.nvm	Encodes length and boost factors for docs and fields
Per-Document Values	.dvd,.dvm	Encodes additional scoring factors or other per-document information.
Deleted Documents	.del	Info about what documents are deleted

Playing around with Solr

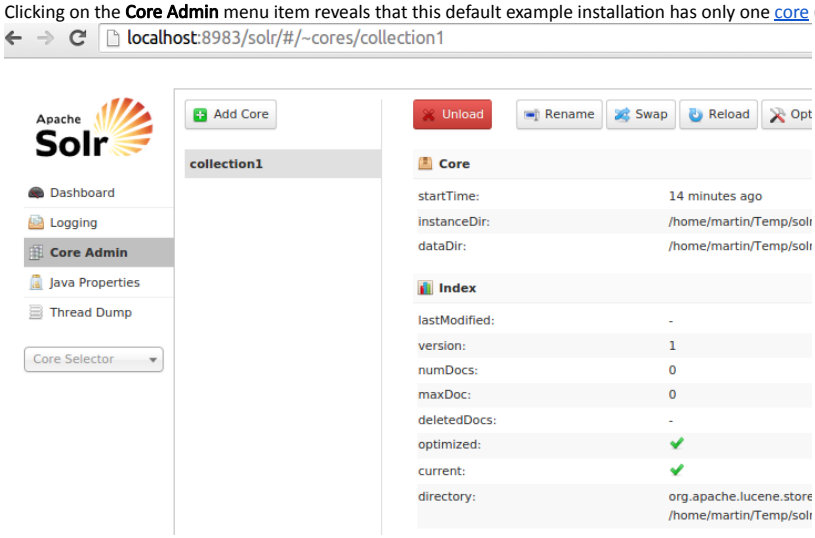
Before we dig into the Alfresco-Solr integration let's play around with a standard Solr installation to get a feel for it. It is quite easy to install Solr. On my Ubuntu box I installed it manually like this (Java is assumed to be already installed):

```
$ wget http://archive.apache.org/dist/lucene/solr/4.10.3/solr-4.10.3.tgz
$ tar -xvf solr-4.10.3.tgz
$ mkdir solr
$ cp -R solr-4.10.3/example solr
Temp$ cd solr/example
Temp/solr/example$ java -jar start.jar
```

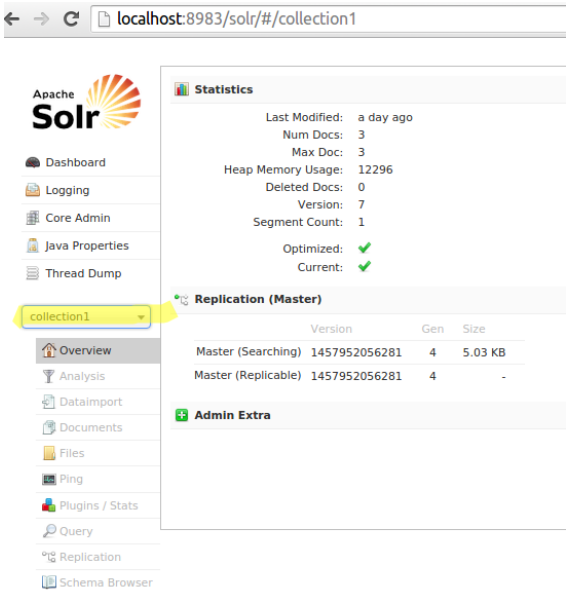
This installation comes with the Jetty application server. We can access Solr Admin Console via http://localhost:8983/solr:



Clicking on the **Core Admin** menu item reveals that this default example installation has only one core (i.e. index) called `collection1`:



For detailed information about a core select the name from the combo box in the lower left corner:



We can see the number of documents in the index etc.
The Solr REST interface has two main access points: the **update URL**, which maintains the index, and the **select URL**, which is used for queries. In the default configuration, they are found at:
http://localhost:8983/solr/update
http://localhost:8983/solr/select

To add a document to the index, we POST an XML representation of the fields to index to the update URL. The XML looks like the example below, with a `<field>` element for each field to index. Such documents represent the metadata and content of the actual documents that we're indexing:

```
<add>
```

```
  <doc>
    <field name="id">CAN5DMARKIII</field>
    <field name="name"> Canon EOS 5D Mark III</field>
    <field name="category">camera</field>
    <field name="features">22.3MP full-frame CMOS sensor</field>
    <field name="features">Canon DIGIC 5+ image processor</field>
    <field name="features">ISO 100 - 25,600</field>
    <field name="features">1080/30p Full HD movie recording</field>
    <field name="features">3.2in, 1040k-dot LCD monitor</field>
    <field name="features">Weather-sealed aluminium chassis</field>
    <field name="weight">950</field>
    <field name="price">3000.00</field>
  </doc>
</add>
```

The `<add>` element tells Solr that we want to add the document to the index (or replace it if it's already indexed), and with the default configuration, the `id` field is used as a unique identifier for the document. Posting another document with the same id will overwrite existing fields and add new ones to the indexed data.

Note that the added document isn't yet visible in queries. To speed up the addition of multiple documents (an `<add>` element can contain multiple `<doc>` elements), changes aren't committed after each document, so we must POST an XML document containing a `<commit>` element to make our changes visible.
Here is an example of how to index the above document with `curl`:

```
$ curl http://localhost:8983/solr/update -H "Content-Type: text/xml" --data-binary '<add>
  <doc>
    <field name="id">CAN5DMARKIII</field>
    <field name="name">Canon EOS 5D Mark III</field>
    <field name="category">camera</field>
    <field name="features">22.3MP full-frame CMOS sensor</field>
    <field name="features">Canon DIGIC 5+ image processor</field>
    <field name="features">ISO 100 - 25,600</field>
    <field name="features">1080/30p Full HD movie recording</field>
    <field name="features">3.2in, 1040k-dot LCD monitor</field>
    <field name="features">Weather-sealed aluminium chassis</field>
    <field name="weight">950</field>
    <field name="price">3000.00</field>
  </doc>
</add>'

<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int name="QTime">63</int></lst>
</response>
```

Then we commit to make the data visible in searches:

```
$ curl http://localhost:8983/solr/update -H "Content-Type: text/xml" --data-binary '<commit />'
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader"><int name="status">0</int><int name="QTime">4</int></lst>
</response>
```

The reason we can add this document to the Solr index is that the example schema (~/Templ/solr/example/solr/collection1/conf/schema.xml) actually contains matching fields:

```
 <field name="id" type="string" indexed="true" stored="true" required="true" multiValued="false" />
  <field name="name" type="text_general" indexed="true" stored="true"/>
  <field name="features" type="text_general" indexed="true" stored="true" multiValued="true"/>
  <field name="weight" type="float" indexed="true" stored="true"/>
  <field name="price"  type="float" indexed="true" stored="true"/>
  <field name="category" type="text_general" indexed="true" stored="true"/>
```

Once we have indexed some data, an HTTP GET on the select URL does the querying. The example below searches for the word "video" in the default search field and asks for the `name` and `id` fields to be included in the response.

```
$ curl "http://localhost:8983/solr/select/?indent=on&q=EOS&df=name&fl=name,id"
<?xml version="1.0" encoding="UTF-8"?>
<response>

<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">7</int>
  <lst name="params">
    <str name="q">EOS</str>
    <str name="indent">on</str>
    <str name="fl">name,id</str>
  </lst>
</lst>
<result name="response" numFound="1" start="0">
  <doc>
    <str name="id">CAN5DMARKIII</str>
    <str name="name">Canon EOS 5D Mark III</str></doc>
</result>
</response>
```

Note that default search field here set to `name`, so if you would have used `q=camera` it would not match anything. To search categories change also the default field (`df`) spec:

```
$ curl "http://localhost:8983/solr/select/?indent=on&q=camera&df=category&fl=name,id"
```

The query language used by Solr is based on Lucene queries, with the addition of optional sort clauses in the query. Asking for `video; inStock asc, price desc`, for example, searches for the word "video" in the default search field and returns results sorted on the `inStock` field, ascending, and `price` field, descending.

> **Note**. in the first article I wrote you could specify default search field in Solr's *schema.xml* configuration file, as in this example:
>
> `<defaultSearchField>text</defaultSearchField>`
>
> This is now **deprecated**, use the `df` parameter as in the above examples instead.

A query can refer to several fields, like `handheld AND category:camera` which searches the `category` field in addition to the default search field.
Besides the `<add>` and `<commit>` operations, `<delete>` can be used to remove documents from the index, either by using the document's unique ID:

```
<delete><id>MA147LL/A</id></delete>
```

Or by using a query to remove a range of documents from the index:

```
<delete><query>category:camera</query></delete>
```

As with add/update operations, a `<delete>` must be followed by a `<commit>` to make the resulting changes visible in queries.

In Solr 3.1 and newer we can also work with JSON as an alternative to XML. So we could add a document as follows to the index using JSON:

```
curl http://localhost:8983/solr/update -H 'Content-type:application/json' -d '
[
{"id" : "book1", "title" : "Apache Solr Enterprise Search Server, Third Edition", "authors" : "David Smiley, Eric Pugh, Kranti Parisa, and Matt Mitchell"},
{"id" : "book2", "title" : "Scaling Big Data with Hadoop and Solr - Second Edition","authors" : "Hrishikesh Karambelkar"}
]'
```

This does not actually work as the `authors` field is not part of the schema:

```
{"responseHeader":{"status":400,"QTime":3},"error":{"msg":"ERROR: [doc=book1] unknown field 'authors'","code":400}}
```

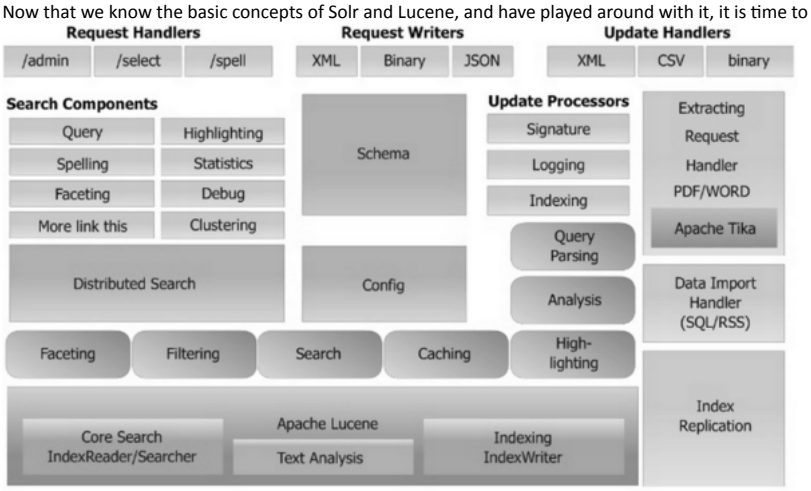
To fix it we can add a field definitions as follows:

```
<field name="authors" type="text_general" indexed="true" stored="true"/>
```

And then restart the server and POST the command again.

Solr Architecture

Now that we know the basic concepts of Solr and Lucene, and have played around with it, it is time to have a look at the architecture:



We have already talked quite a bit about Apache Lucene that is the foundation building block for Solr, providing the core indexing and search functionality. On top of Lucene we have specific search feature implementations such as faceting, filtering, highlighting, spelling etc. The Solr Schema and Configuration components control most of the functionality around indexing and searching.

To obtain maximum query performance, Solr stores several different pieces of information using in-memory caches. Result sets, filters and document fields are all cached so that subsequent, similar searches can be handled quickly.

Solr is a web application and it provides a REST-based interface that can be used to access most of the features of the Solr search server. The REST interface is implemented as so called request handlers that are used to query the index and update handlers that are used to update the index.

Apache Tika is built into Solr to support extracting plain text from file formats such as PDF and MS Word. Making the document text ready for indexing.

The conceptual architecture diagram also contains features such as index replication and distributed search that we will talk more about in the "*Searching with Alfresco 5.1 Enterprise*" article.

How is Solr integrated with Alfresco?

Okay so we have now looked at Solr by itself. Let's see how it is integrated with Alfresco. The easiest way to put it is as follows – Alfresco uses HTTP GETs to talk to Solr and search for content. Solr updates the indexes by talking to Alfresco to get the number of transactions that have been committed since it last talked to Alfresco. The custom components in Solr that talks to Alfresco are referred to as trackers.

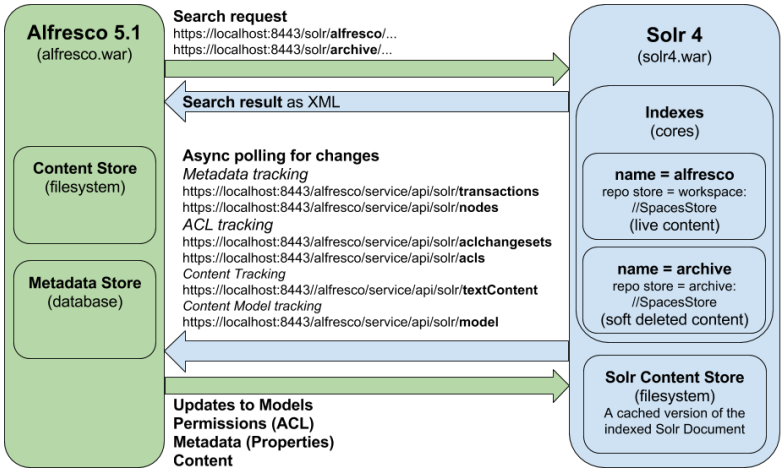
Solr talks to Alfresco every 15 seconds to see if any new transactions have been committed. So the index will eventually be up to date with the database. This means that if you upload a document to Alfresco and then immediately execute a full text search, it's not 100% sure that you will hit the document just uploaded. An important thing to know is that unlike with Solr 1, Solr 4 indexing is split between metadata tracking, ACL tracking, and content tracking.

Solr will also keep track of new custom content models that have been deployed and download them to be able to index the properties in these models. Any permission settings will also be downloaded by Solr so it can do query time permission filtering.

There are 2 cores (i.e. indexes):

- **WorkspaceStore**: for searching all live content (i.e. stuff in alf_data/contentstore)
- **ArchiveStore**: for searching soft deleted content (i.e. content in alf_data/contentstore that has been marked as deleted)

The following picture gives you an overview of how it works:



As the picture shows, Solr version 4 is used and has been extended with code to talk to Alfresco. So if you are going to install Solr on a separate dedicated search box you need to download it from Alfresco download site and not from Apache (more information about this later on).

> If you wanted to use a newer version of Solr such as for example 5.5, then that would be a bit of a problem. It would not be a simple task to upgrade and patch it with Alfresco specific code for trackers etc, and Alfresco might not support you after the upgrade.

Note the new Solr Content Store that has been added to store every indexed Solr Document, including metadata and content text (new for the Alfresco 5 - Solr 4 integration). This way things like re-indexing and updates can be done in a quicker way. This has nothing to do with stored document fields, which are not used much at all by the Alfresco Solr Schema. Only the `id`, `DBID`, and `_version_` (internal field that is used by the partial update procedure - see the *Searching with Alfresco 5.1 Enterprise* article) fields are stored. And if you define a new property in a content model with index attribute stored set to true, this will have no effect on what's returned in search results. To return all fields (i.e. the complete document that was indexed) see the following section.

So when Solr is used to search for content, how is it affecting Alfresco Share if for example the Solr server stops working or goes down? It's **not** affecting the normal browsing and navigating around in the Alfresco Share UI, as this is not using Solr but instead regular database queries.

Solr is used in the following situations:

- Instant Search (search field in top right corner)
- Advanced Search
- Filters
- Tags
- Categories (implemented as facets)
- Dashlets such as the Recently Modified Documents, Site Content
- People Finder and Site Finder pages
- Wildcard searches for People, Groups, Sites (uses database search if not wildcard)

The following properties in **alfresco-global.properties** are related to Solr, and they are setup as follows by default:

```
### Solr indexing ###
index.subsystem.name=solr4
dir.keystore=${dir.root}/keystore
solr.host=localhost
solr.port.ssl=8443
```

As you can see, the Solr 4 search configuration is contained in a subsystem implementation with the name `solr4`, there is also an implementations for Solr 1.4 (`solr`).

There are more properties available to configure if we look in the main configuration file for the solr4 subsystem implementation. This file is called **solr-search.properties** and is located in the **alfresco/subsystems/Search/solr4** directory in the **alfresco-repository-5.1.e.jar** file. Yes, that's right, most of the configuration files these days are located inside JAR files. In previous versions of Alfresco you could find most default values for properties in the **repository.properties file located in the alfresco/tomcat/webapps/alfresco/WEB-INF/classes/alfresco** directory in the exploded web application.

Here are some of the other properties (copy them to alfresco-global.properties if you need to change them):

```
solr.host=localhost
solr.port=8083
solr.port.ssl=8446
solr.query.includeGroupsForRoleAdmin=false
solr.query.maximumResultsFromUnlimitedQuery=${system.acl.maxPermissionChecks}
solr.baseUrl=/solr4

#
# Solr Suggester properties
#
solr.suggester.enabled=true
```

These properties are mostly related to how Alfresco connects to the Solr server, and as it is running in the same Tomcat instance as Alfresco, the connection properties will be setup to connect to a locally running Solr server. Alfresco will also by default use HTTPS to connect to the Solr server (for information about how to use plain HTTP see here). Note that the URL for accessing the Solr web application is now `/solr4` compared to previously when it was `/solr`.

A couple of the properties need a bit of extra explanation:

- `solr.suggester.enabled` - if set to `true` then Alfresco uses the suggester component in Solr to provide users with automatic suggestions for query terms.
- `solr.query.includeGroupsForRoleAdmin` - if set to `true` then search results will include any groups associated with ROLE_ADMINISTRATOR, if `false` then no groups will be returned for ROLE_ADMINISTRATOR
- `solr.query.maximumResultsFromUnlimitedQuery` - this controls the maximum number of items that will be returned in the search result set. By default this is set to `system.acl.maxPermissionChecks` to align with maximum number of access control list (ACL) permission checks that will be done for a search result to determine if the requesting user are allowed to read a node or not. This is 1000 by default. Practically this means that you will never see more than 1000 nodes (folders, files, etc) in the search result in Alfresco Share, or from any other interface.

There are also some properties associated with Solr Sharding, which we will cover in the "*Searching with Alfresco 5.1 Enterprise*" article.

Alfresco Solr Request Handlers

The request handlers manage search requests coming into Solr and there are some Alfresco specific request handlers that you will see if you open up the <alfrescoinstalldir>/solr4/<core instanceDir>/conf/**solrconfig.xml** for a core:

- **/afts** - Alfresco Full Text Search (FTS) request handler with spell checking, faceting, highlighting, and clustering. It is named based on the query handler that is part of the component list. And the query handler supports the extended Lucene query language called Alfresco Full Text Search (FTS).
- **/alfresco** - Similar to the /afts handler except it does not provide spell checking, clustering, and the custom Alfresco FTS syntax. Only the standard Lucene query syntax is supported.
- **/cmis** - Support for CMIS-QL

Alfresco Solr Update Handlers

There are no specific update handlers used in the Alfresco-Solr integration as Solr is customized to call back to Alfresco Repository for content and metadata that should be indexed.

Term Positions

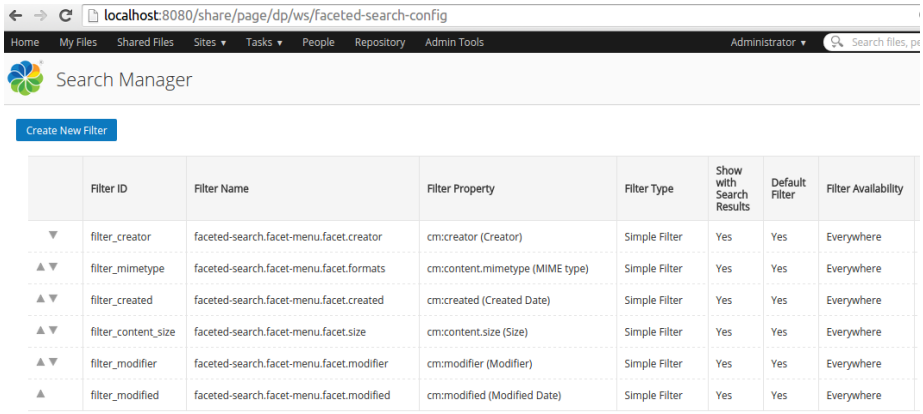
All term positions are stored but used as little as possible because it is expensive during query time. They are used when doing re-ranking of search result.

Faceting

Facets are support for each search result and a number of facets are configured and available out-of-the-box:

- Creator (`cm:creator`)
- Created Date (`cm:created`)
- Modifier (`cm:modifier`)
- Modified Data (`cm:modified`)
- Content Size (`cm:content` size attribute)
- Content Mimetype (`cm:content` size mimetype attribute)

Here is a screenshot from the Search Manager in Alfresco Share, showing the facet configuration:



	Filter ID	Filter Name	Filter Property	Filter Type	Show with Search Results	Default Filter	Filter Availability
▼	filter_creator	faceted-search.facet-menu.facet.creator	cm:creator (Creator)	Simple Filter	Yes	Yes	Everywhere
▲▼	filter_mimetype	faceted-search.facet-menu.facet.formats	cm:content.mimetype (MIME type)	Simple Filter	Yes	Yes	Everywhere
▲▼	filter_created	faceted-search.facet-menu.facet.created	cm:created (Created Date)	Simple Filter	Yes	Yes	Everywhere
▲▼	filter_content_size	faceted-search.facet-menu.facet.size	cm:content.size (Size)	Simple Filter	Yes	Yes	Everywhere
▲▼	filter_modifier	faceted-search.facet-menu.facet.modifier	cm:modifier (Modifier)	Simple Filter	Yes	Yes	Everywhere
▲	filter_modified	faceted-search.facet-menu.facet.modified	cm:modified (Modified Date)	Simple Filter	Yes	Yes	Everywhere

Suggester

The Solr Suggester Component is used to enable search query suggestions.

Auto-Phrasing

When a user searches for: *the big red banana*, what they really want to see at the top in the search result is docs matching the phrase: "*the big red banana*". This is auto phrasing. However, searching for phrases is really slow as you have to use term position. Phrases become a bigger and bigger problem with large repositories. So Alfresco uses re-ranking instead.

Re-Ranking

Doing auto-phrasing is expensive so after you get the initial search result do a re-rank of search result with phrase taken into account.

Multilingual Search

The Alfresco - Solr integration supports searching in a specific locale, which is specified in the search query as follows:

```
:/solr4/alfresco/afts?wt=json&fl=DBID%2Cscore&rows=25&df=keywords&start=0&locale=en_GB&...
```

It is also possible to auto-detect local used in the query string by adding properties to the Alfresco FTS Query Parser plugin configuration, located in the **solrconfig.xml** file:

```
<queryParser name="afts" class="org.alfresco.solr.query.AlfrescoFTSQParserPlugin">
  <str name="rerankPhase">QUERY_PHASE</str>
  <str name="autoDetectQueryLocale">true</str>
  <str name="autoDetectQueryLocales">en,fr</str>
  <str name="fixedQueryLocales">de</str>
</queryParser>
```

In this example we have turned on auto detection of locales in the query string (it is turned off by default). So if the query string looked like this:

```
:/solr4/alfresco/afts?wt=json&fl=DBID%2Cscore&rows=25&df=keywords&start=0&locale=en_GB&....q=Göteborg&...
```

Locale auto detection would discover Swedish text (i.e. `Göteborg`) in the query string and add locale `sv`. So Solr will search both the `text_en` and `text_sv` fields. And possibly discover the text `Göteborg` in both English and Swedish content. In fact, because we have also specified the **fixedQueryLocales** property to `de`, the search will also (always) include German content via the `text_de` field.

There are basically two ways you can implement multilingual search in Solr:

- Using a separate field per language
- Using a separate core per language

Alfresco uses the first approach with a separate field per language. So in the **schema.xml** file you will see text field definitions like this:

```
<!-- English -->
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
<!-- Arabic -->
<fieldType name="text_ar" class="solr.TextField" positionIncrementGap="100">
<!-- Bulgarian -->
<fieldType name="text_bg" class="solr.TextField" positionIncrementGap="100">
<!-- Catalan -->
<fieldType name="text_ca" class="solr.TextField" positionIncrementGap="100">
<!-- Kurdish -->
<fieldType name="text_ckb" class="solr.TextField" positionIncrementGap="100">
<!-- Czech -->
<fieldType name="text_cz" class="solr.TextField" positionIncrementGap="100">
<!-- Danish -->
<fieldType name="text_da" class="solr.TextField" positionIncrementGap="100">
```

The above field definition is for the content text, it is also possible to search on multilingual metadata. In the schema this is supported by the following type of dynamic field definitions:

```
dynamicField name="mltext@m__l_@*"  type="alfrescoFieldType" indexed="true" omitNorms="true"   stored="false"  multiValued="true" />
```

As you probably know, the content model supports multilingual properties via the d:mltext data type. A good example out-of-the-box of this is the `cm:titled` aspect, which contains both a multilingual title and description property.

Important: Alfresco Share version 5- does not support managing multilingual content at the moment. So to test it we would need to add multilingual content and metadata programmatically.

Virtual Schema

Alfresco supports a dictionary of types, aspects, and properties that can be updated dynamically. For example, by adding a new property. The dictionary specifies how properties can be used at query time. Text may be treated as an identifier, used for full text search, or both. All property types may be single or multi-valued and may be sorted or faceted with varying degrees of support (No support, Lucene Field Cache, or Lucene DocValues).

At index time, the virtual schema is used to map Alfresco content model properties to one or more dynamic fields in the Solr schema. Here are examples of dynamic field definitions:

```
<dynamicField name="int@s_@*"         type="int"          indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="long@s_@*"        type="long"         indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="float@s_@*"       type="float"        indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="double@s_@*"      type="double"       indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="date@s_@*"        type="date"         indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="datetime@s_@*"    type="date"         indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="false" sortMissingFirst="false" />
<dynamicField name="boolean@s_@*"     type="oldStandardAnalysis" indexed="true" omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="true" />
```

At query time, one or more fields may be used for a query, and the best available field will be selected for faceting and sorting.

The virtual schema is also used for other things, such as generating queries for type hierarchies.

Full Text Properties

A custom field type is used to select a locale specific analyser for text properties and multilingual text properties, here are some examples:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
<fieldType name="text_de" class="solr.TextField" positionIncrementGap="100">
<fieldType name="text_es" class="solr.TextField" positionIncrementGap="100">
```

The selected analyser is wrapped to generate localised tokens that can be mixed in the same lucene index field. Text may be indexed in more than one field, for example for:

- Full Text Search (FTS)
- Identifier and Pattern Based Search
- Sorting
- Faceting

Text properties have custom support for localised ordering. Multilingual properties have additional support to select the most appropriate value from the locales available to sort a given document. This means that if you have documents that have only French or German titles they can be ordered correctly in English, Finnish etc.

Some text properties, typically just the document name, can be configured to support additional non localised queries. File names can follow patterns and naming conventions that are independent of locale. For example, this support allows files named “BigRed-Dangerous.dog” to be found by “Big” “Red” “Dangerous dog” etc. (This uses the Solr word delimiter factory). Queries for name will include localised matches and non-localised.

Queries can also be generated for more than one locale - to match German or French titles with the appropriate stemming.

Alfresco Repository Tracking

Several trackers run and pull information from Alfresco independently. All cores build their index independently from each other. Tracking/pull allows us to do bulk and parallel updates and requests.

Model Tracking

Content Models are pulled periodically from Alfresco and cached locally. This supports dynamic changes to models and static configuration of models in Alfresco. Some model changes are disallowed as they would break the current lucene index.

Metadata Tracking

Pulling all document information except content in the order in which it was last modified. This will be pre-filtered by ACL ID when sharded. The changes are committed in transactional chunks. This includes moving documents between shards when ACLs change. Metadata tracking marks content as out of date but may index the last cached content until it is updated by the Content Tracker.

Metadata tracking takes care of rebalancing nodes if their ACL changes as part of a move or adding a permission to a node that previously had no permissions.

ACL Tracking

Pulling all Access Control List (ACL) information in the order in which it was last modified. This will be pre-filtered by ACL ID when sharded. (ACLs never change their ID so can never move between shards)

Content Tracking

Update content that is out of date. Metadata is updated more aggressively. It is OK to have out of date content. Benchmarking tests has showed metadata updates keeping up at a constant 100/s up to 50M docs. Content tracking depends on transformation time.

Cascade Update of PATH

When documents move or are renamed it affects their paths and the paths to all of their descendents. Renaming and moving a document may affect the paths to documents in other shards. These are updated too. This is currently done as part of metadata indexing but it is split out in 5.1+.

Reindex by Query

Reindex a subset of documents in the index. This can be used to support limited model/schema changes that affect a small number of documents. For example to reindex a property for faceting or to be used as an identifier. Currently not multi-threaded (multi-threaded in 5.1+).
A deferred schema change (one that would affect how stuff is indexed and break query) can be allowed in a controlled way. The schema update is allowed. The information held in the index will be "wrong" until reindexed to match the current schema. The data is reindexed by query. This can remove the need to do a full reindex for some model and implied schema changes.

Document Content Store

Cache each Solr document we add to the index (also, see above figure), this has several benefits:

- Speeds index rebuild - locally cached
- Metadata only changes can reuse cached content
- Index metadata first and add content later (low latency metadata)
- Document decoration - all fields
- Highlight any field (5.1+)
- Use old content when indexing metadata until content is fixed up later
- Eventually consistent read only copy of indexed nodes

Security Enforcement

Here are some of the features backing security checks on returned search results:

- Query time enforcement of access control - allow, deny and ownership
- Filters for security markings (Records Management)
- Facets and total counts reflect user permissions
    - This is not possible without this type of integration
- Co-mingle documents and authorization information in the same Lucene index
    - In memory, DocValue based filter/JOIN (and post filter in 5.1+)

Custom Result Groupings

Result Grouping groups documents with a common field value into groups and returns the top documents for each group. This can be done on `mimetype`. For applications of this see this blog entry.

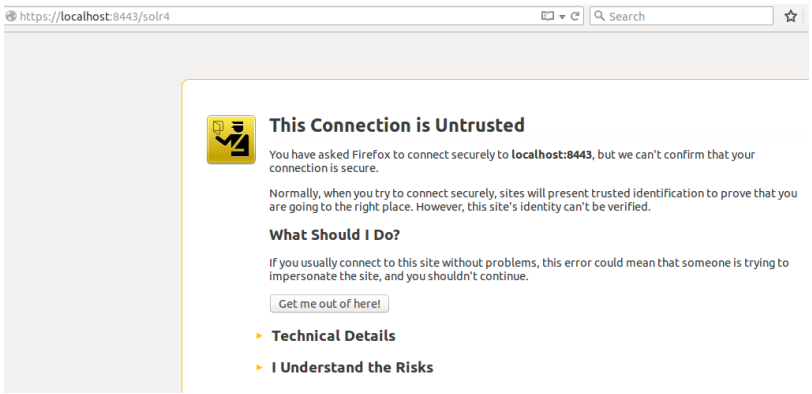
Solr Core Configuration Templates

These are tested Solr configurations to build shard instances. Can be the base for customer extensions. Can alter locale specific tokenisation, synonyms, stopwords, etc.
You can find the templates in the **<alfrescoinstalldir>/solr4/templates** directory.

Logging into Alfresco Solr Admin UI

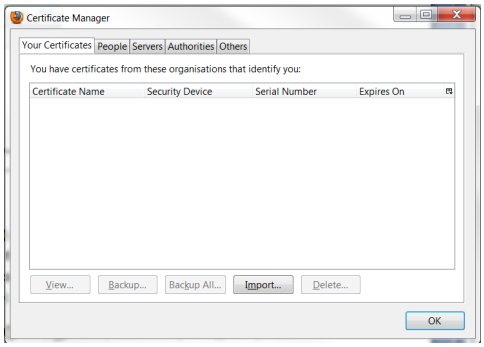
The Solr web application comes with an administration UI that can be useful for finding out stuff about the Solr installation, such as deployed schemas, Solr configuration, indexed fields etc. The admin console can be accessed via the following URL:

https://localhost:8443/solr4
The first time you hit this URL it will respond with an error message as follows:

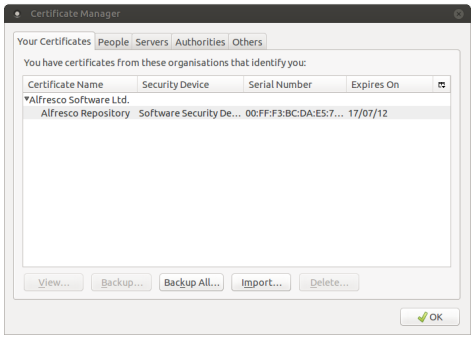


All URLs for the Solr 4 web application (i.e. /solr4) are protected by SSL. In order to use these from a browser you need to import a browser-compatible keystore to allow mutual authentication and decryption to work. Follow these steps to import the keystore into your browser (I'm using Firefox, other browsers will have a similar mechanism):

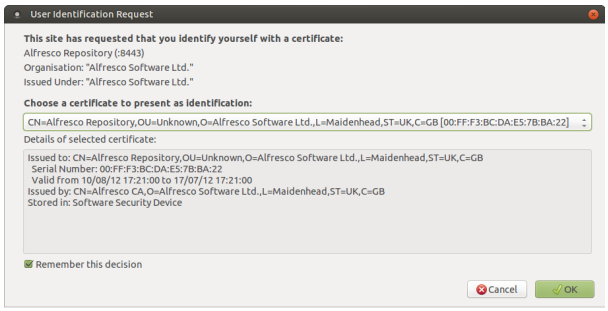
1) Open the Firefox Certificate Manager (**Edit | Preferences | Advanced | Certificates | View Certificates | Your Certificates**):



2) Import the browser keystore **browser.p12** that is located in your **<alfrescoinstalldir>\alf_data\keystore** directory. You will be prompted for password. The password is `alfresco`. This should result in a dialog indicating that the keystore has been imported successfully.
3) The Certificate Manager should now show one imported certificate as follows:

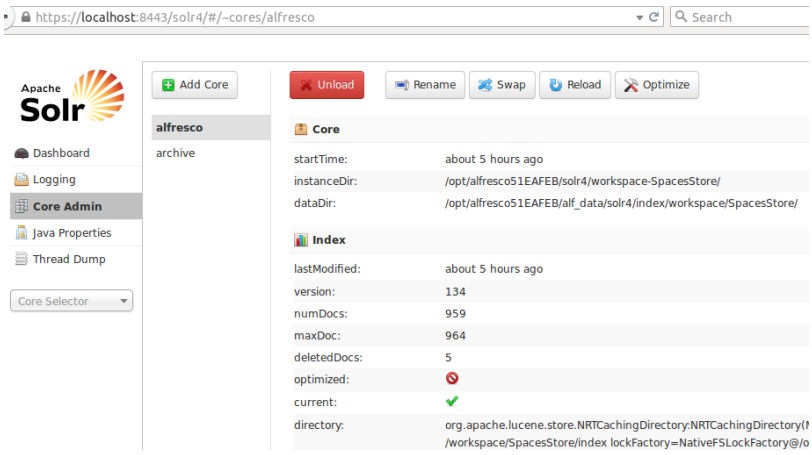


4) Now close the Certificate Manager by clicking **OK**
5) Then hit https://localhost:8443/solr4 again, this will show the following dialog (you might have to restart FF):



6) The Solr web application now wants you to identify (i.e. authenticate) yourself with a certificate for access to the Solr admin console. Select the Alfresco Repository cert and click **OK**. This means that we are using the same certificate as the Alfresco Repository is using when it talks to the Solr web application during for example a search.
7) We should now see the Admin Home page:

8) Click on the **Core Admin** menu item and two cores should be available to view, with the live content in the `alfresco` core displayed:



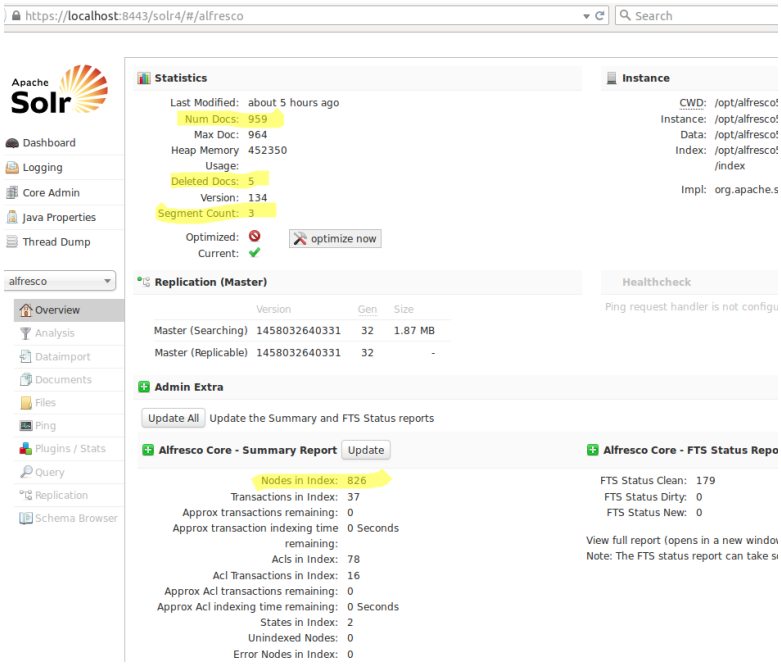
Click on the **archive** item in the Core list to see information about the Archived nodes index in Alfresco (i.e. soft deleted nodes).

Working with the Alfresco Solr Admin UI

The Solr Admin features constantly improves and it is worth having a look at what's available in each new version.

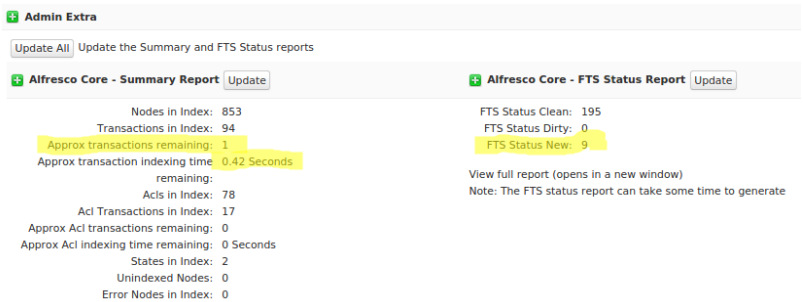
Core Overview Page

To see an overview of a core use the **Core Selector** combo box in the lower left corner of the screen, here is the information for the `alfresco` core with the live content index:



We can see how many documents that are in the index (i.e core), how many documents have been deleted, how many segments we got etc. Some information is Alfresco specific (i.e. you would not see it in the standard Solr Admin UI) such as the "Alfresco Core -" sections.

If you upload a large number of files to Alfresco the **Alfresco Core -** reports will tell you if Solr has finished indexing metadata and content for them or not, which means they will appear in search results:



The **Alfresco Core - Summary Report** shows the metadata transaction tracking status (Alfresco tracking in Solr 4 has been split up into metadata transaction tracking and content tracking). In the above example we can see that there is one metadata transaction left to index. We can also see how long Solr thinks it is going to take before it has finished indexing all remaining Alfresco metadata transactions.

To the right is the **Alfresco Core - FTS Status Report**, which informs us that there are 9 content files left to index, if text can be extracted from them. Full Text Search (FTS) will only work when content indexing is complete for a file.

When doing a re-indexing of an entire repository the Core Overview page can be useful to get an understanding of how much is left to do. Take the following example:

Metadata transaction tracking status:

```
Acls in Index:                                  232997
Nodes in Index:                                  9774798
Transactions in Index:                          816762
Approx transactions remaining:                  0
Approx transaction indexing time remaining:      0 Seconds
Acl Transactions in Index:                         60992
Approx Acl transactions remaining:           0
Approx Acl indexing time remaining:           0 Seconds
States in Index:                                   2
Unindexed Nodes:                                   0
Error Nodes in Index:                         0
```

Content Tracking status:7921438

```
FTS Status Clean:        2745518
FTS Status Dirty:            71125
FTS Status New:           5175920
```

Here it is clear that the metadata transaction tracking is complete and index is up to date with the latest database transaction. The indexing of the content files is however not complete, and there is approximately 65% left of the content to index, if the average file size of the remaining files are similar to the ones that have already been indexed.
FTS Status Dirty means that cached content is out-of-date. *FTS Status New* means that the content has not been previously indexed and is not in the cache. *FTS Status Clean* means that content index is up-to-date and cache is in sync.
If content tracking is taking a very long time it is worth checking how many content files that are processed in each batch by Solr. This is controlled by the following properties for each core (<alfrescoinstalldir>/solr4/workspace-SpacesStore/conf/solrcore.properties):

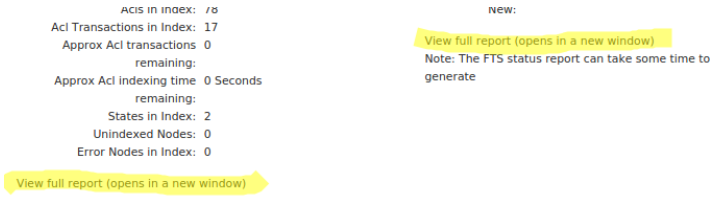
```
# Batch fetch
alfresco.contentReadBatchSize=4000
alfresco.contentUpdateBatchSize=1000
```

If these properties have very low values then it might be worth increasing the batch size and see if re-indexing of content goes faster. The higher the values the fewer queries and commits. The values above are from a standard 5.1.e installation and should be sufficient for most scenarios.
There are other properties for controlling the batch fetch size for metadata:

```
alfresco.transactionDocsBatchSize=500
alfresco.nodeBatchSize=100
alfresco.changeSetAclsBatchSize=500
alfresco.aclBatchSize=100
```

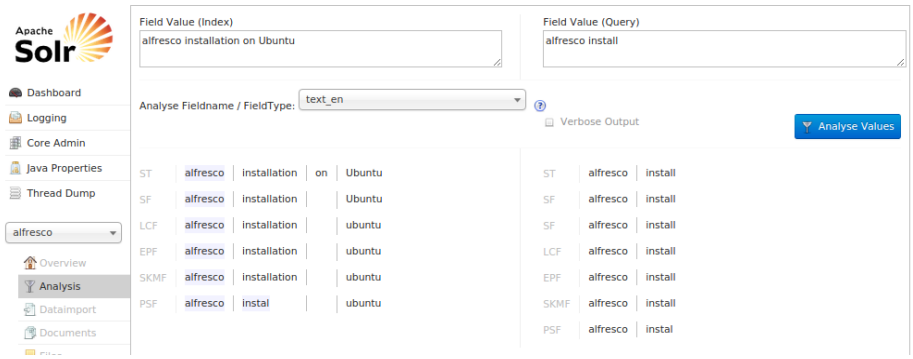
For more information about these properties see this configuration section.

If you need all the information you can get around metadata indexing and content indexing you can turn to the full reports, which can be accessed via links under the summary reports:



Core Analysis Page

The Analysis page lets you inspect how data will be handled according to the field, field type and dynamic rule configurations found in **<alfrescoinstalldir>/solr4/<core>/conf/schema.xml**. You can analyze how content would be handled during indexing or during query processing and view the results separately or at the same time. Ideally, you would want content to be handled consistently, and this screen allows you to validate the settings in the field type or field analysis chains.
Enter content in one or both boxes at the top of the screen, and then choose the field or field type definitions to use for analysis:



In this case we have chosen to analyze the `text_en` field in the schema. To see how English text is handled during indexing time we have supplied “alfresco installation on Ubuntu” in the `Field Value (index)` box and to see how text is handled during a search (i.e. query) we have put “alfresco install” in the `Field Value (query)` box.
We can see a number of filters invoked in the indexing and query analysis chain:

- `ST`: Standard Tokenizer breaking up the text stream into words/terms/tokens
- `SF`: Stop Word filter that removes stopwords such as “on”
- `LCF`: Lowercase filter that converts term to lowercase
- `EPF`: English Possessive Filter that removes possessives (trailing 's) from words
- `SKMF`: Keyword Marker filter that removes terms present in the **protwords.txt** file

- PSF: Porter stemming filter

The query analysis chain also contains a synonym filter (SF) that will look for synonyms in the **synonyms.txt** file.

These analysis chains corresponds exactly to what has been configured in the **schema.xml** for the field:

```
<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt" />
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true" expand="true"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt" />
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EnglishPossessiveFilterFactory"/>
    <filter class="solr.KeywordMarkerFilterFactory" protected="protwords.txt"/>
    <filter class="solr.PorterStemFilterFactory"/>
  </analyzer>
</fieldType>
```

So if you are searching and not hitting anything, then this tool might be useful to help figuring out why a specific query is not matching anything.

Core Data Import Page

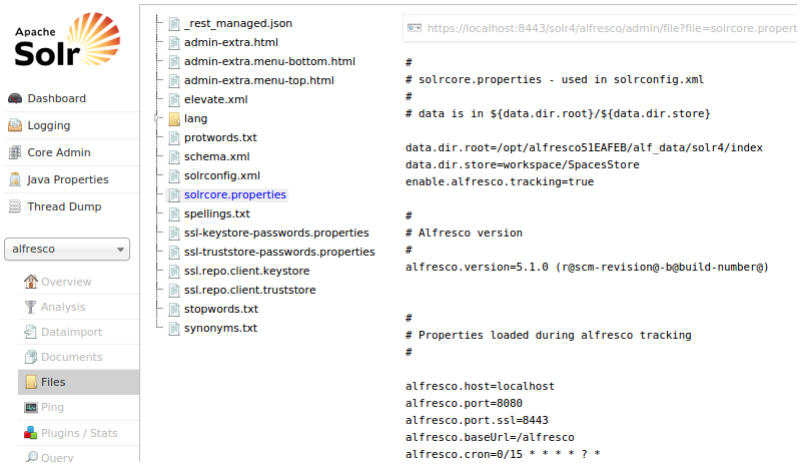
There is no data import handler defined, all data comes from querying Alfresco repository, so skip this page.

Core Documents Page

This page can be used to submit Solr Documents to update the index. This is not so easy to do with the Alfresco-Solr integration as you would have to put together quite a complex Document and submit it. The Alfresco - Solr integration does not use the Solr update handlers but instead custom Solr code (trackers) pull data from Alfresco.

Core Files Page

This page is really useful if you don't have access to the box where Alfresco Solr is running. You can then use this screen to view the content of each configuration file for the specific core:

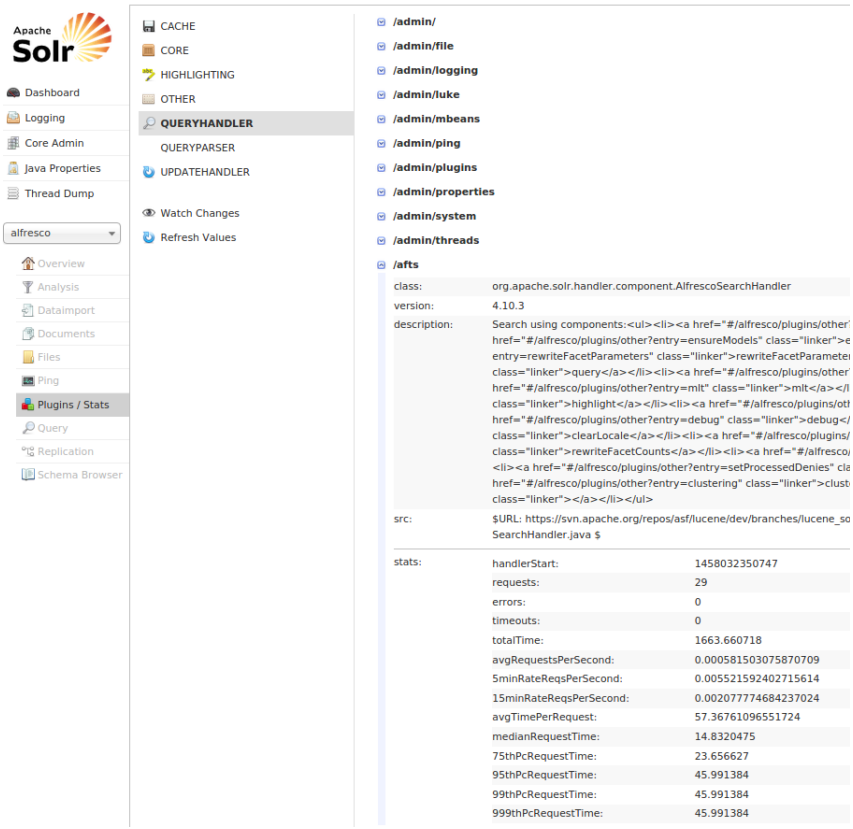


Core Ping Page

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

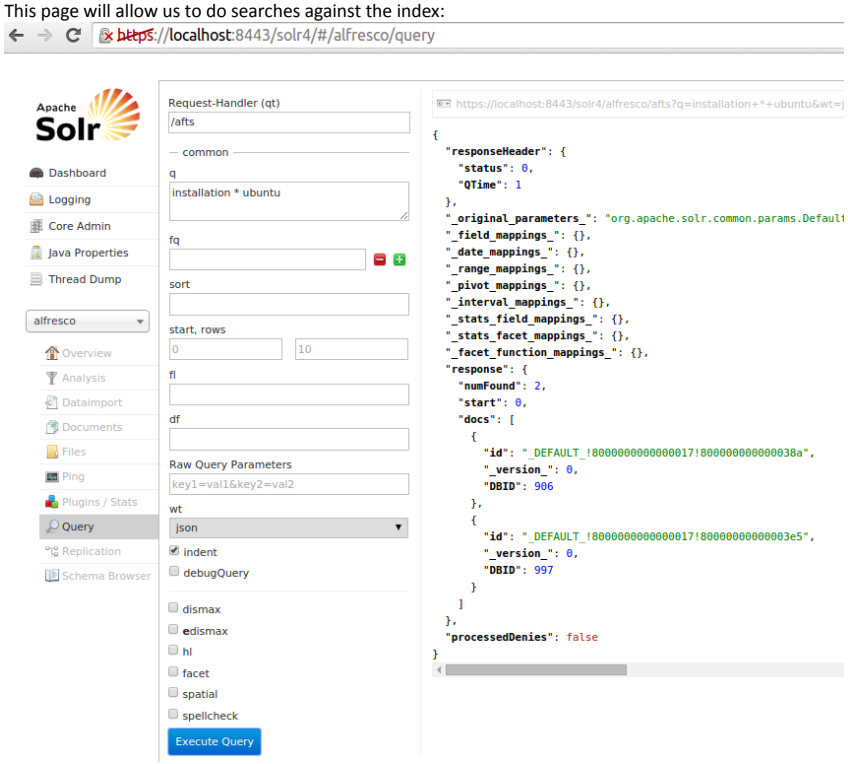
Core Plugin Page

This page gives access to information and statistics for all the handlers used for indexing (update) and querying:

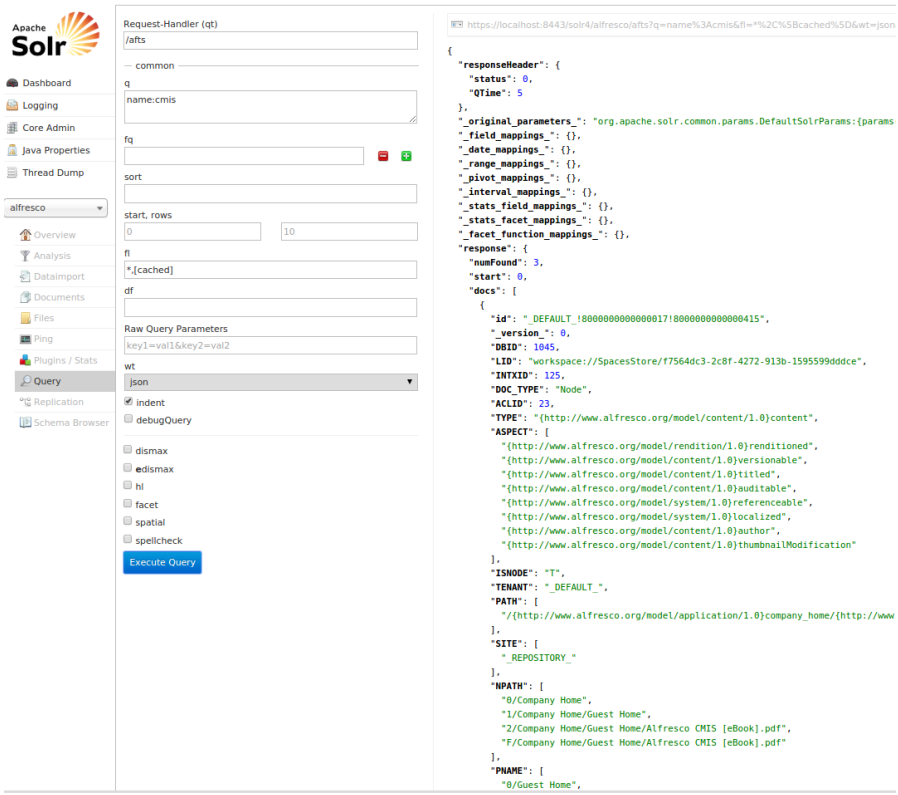


Core Query Page

This page will allow us to do searches against the index:



Here we are doing a proximity search with the query "*installation * ubuntu*" and get 2 results back. Note that there are not that many fields that are actually stored and returned in the search result. Only the document ID, database ID, and version. We would need to make a database call and get the rest of the properties via the DBID.
However, because the complete Solr document that was indexed is stored in the custom Solr content store there is a way to return it in the search result. It is not possible to select specific fields but you can get the complete document:



The way to do it is to use the special field "`[cached]`" in the `fl` input box. Note that you cannot use it alone, it has to be specified as "`*,[cached]`". The returned result also provide important information about how each content model property has been mapped to schema field. Scroll down a bit and you should see a field called `FIELDS`:

```
"FIELDS": [
  "boolean@s_@{http://www.alfresco.org/model/content/1.0}autoVersionOnUpdateProps",
  "datetime@sd@{http://www.alfresco.org/model/content/1.0}created",
  "text@m__lt@{http://www.alfresco.org/model/content/1.0}lastThumbnailModification",
  "text@m___t@{http://www.alfresco.org/model/content/1.0}lastThumbnailModification",
  "text@s__lt@{http://www.alfresco.org/model/content/1.0}creator",
  "text@s___t@{http://www.alfresco.org/model/content/1.0}creator",
  "text@s__l_@{http://www.alfresco.org/model/content/1.0}creator",
  "text@sd___@{http://www.alfresco.org/model/content/1.0}creator",
  "text@s__lt@{http://www.alfresco.org/model/system/1.0}node-uuid",
  "text@s___t@{http://www.alfresco.org/model/system/1.0}node-uuid",
  "text@s__lt@{http://www.alfresco.org/model/system/1.0}store-protocol",
  "text@s___t@{http://www.alfresco.org/model/system/1.0}store-protocol",
  "text@s__lt@{http://www.alfresco.org/model/content/1.0}name",
  "text@s___t@{http://www.alfresco.org/model/content/1.0}name",
  "text@s__l_@{http://www.alfresco.org/model/content/1.0}name",
  "text@s____@{http://www.alfresco.org/model/content/1.0}name",
  "text@s__sort@{http://www.alfresco.org/model/content/1.0}name",
  "suggest_@{http://www.alfresco.org/model/content/1.0}name",
  "long@s_@{http://www.alfresco.org/model/system/1.0}node-dbid",
  "text@s__lt@{http://www.alfresco.org/model/system/1.0}store-identifier",
  "text@s___t@{http://www.alfresco.org/model/system/1.0}store-identifier",
  "locale@s_@{http://www.alfresco.org/model/system/1.0}locale",
  "text@s__lt@{http://www.alfresco.org/model/content/1.0}versionLabel",
  "text@s___t@{http://www.alfresco.org/model/content/1.0}versionLabel",
  "text@sd___@{http://www.alfresco.org/model/content/1.0}versionLabel",
  "text@s__lt@{http://www.alfresco.org/model/content/1.0}modifier",
  "text@s___t@{http://www.alfresco.org/model/content/1.0}modifier",
  "text@s__l_@{http://www.alfresco.org/model/content/1.0}modifier",
  "text@sd___@{http://www.alfresco.org/model/content/1.0}modifier",
  "datetime@sd@{http://www.alfresco.org/model/content/1.0}modified",
  "boolean@s_@{http://www.alfresco.org/model/content/1.0}autoVersion",
  "boolean@s_@{http://www.alfresco.org/model/content/1.0}initialVersion",
  "content@s__lt@{http://www.alfresco.org/model/content/1.0}content",
  "content@s___t@{http://www.alfresco.org/model/content/1.0}content",
  "suggest_@{http://www.alfresco.org/model/content/1.0}content"
],
```

So here we can see that for example the `cm:created` property was mapped to the `datetime@sd@` dynamic field:

```
<dynamicField name="datetime@sd@*"  type="date"  indexed="true" omitNorms="true"  stored="false"  multiValued="false"  docValues="true" sortMissingLast="false" sortMissingFirst="false" />
```

Dynamic fields follow a naming convention, for example "datetime@sd@" means datetime data type, s = single value, d = docValue.

A property can have multiple field mappings to support different things like faceting and tokenization.

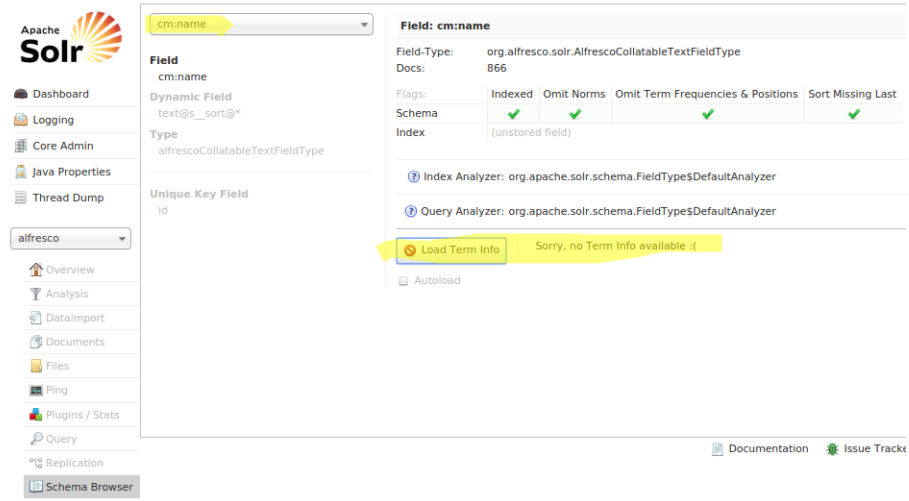
The stored Solr document also provides other important information about the content:

```
"cm:content.docid": 284,
"cm:content.size": 4012333,
"cm:content.locale": "en_GB",
"cm:content.mimetype": "application/pdf",
"cm:content.encoding": "UTF-8",
"sys:store-identifier": "SpacesStore",
"sys:locale": "en_GB_",
"cm:versionLabel": "1.0",
"cm:modifier": "admin",
"cm:modified": "2016-03-15T15:04:06.539Z",
"cm:autoVersion": "true",
"cm:initialVersion": "true",
"cm:content.tr_status": "org.alfresco.solr.client.SOLRAPIClient$SolrApiContentStatus:OK",
"cm:content.tr_ex": null,
"cm:content.tr_time": 6340,
"cm:content": [
  "\n\n\nAlfresco CMIS\n\nEverything you need to know to start coding \nintegrations with a
],
"FTSSTATUS": "Clean"
```


Here we can find out the size of the content, the locale, MIME-type etc. There is also information about the transformation of content to text in the `cm:content.tr_*` properties. They will tell us if the content has been transformed successfully to text and how long it took. Any exceptions would be found in the `tr_ex` property, and it would then not be possible to do full text search. The `FTSSTATUS` property also tells you if the index for this content is up-to-date with the content in Alfresco Repository, `Clean` means that everything is indexed and up-to-date (`Dirty` = needs to be updated, `New` = not yet indexed).

Core Schema Browser Page

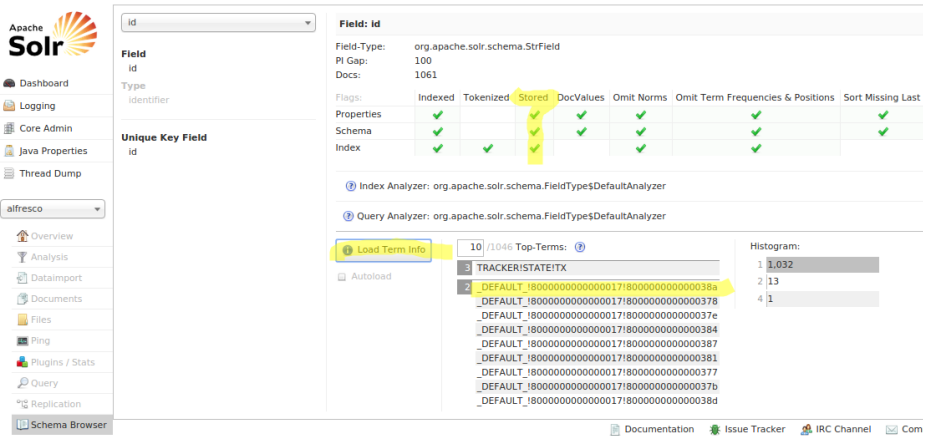
The last core page is the **Schema Browser** and it looks like this:



Here I have selected the familiar `cm:name` property from the Alfresco general content model. This property is here represented as a dynamic field `text@__sort@*`. Note that it is not stored and therefor not returned in search results. Note also that a property such as `cm:name` will have multiple field mappings and the `text@__sort@*` mapping is just for when the field is used in sorting scenarios.
A full list of field mappings for `cm:name` looks like this:

- text@s__lt@{http://www.alfresco.org/model/content/1.0}name = **l**ocalized and **t**okenized **s**ingle value
- text@s___t@{http://www.alfresco.org/model/content/1.0}name = **t**okenized **s**ingle value
- text@s__l_@{http://www.alfresco.org/model/content/1.0}name = **l**ocalized and **non**-tokenized **s**ingle value
- text@s____@{http://www.alfresco.org/model/content/1.0}name = **s**ingle value
- text@s__sort@{http://www.alfresco.org/model/content/1.0}name = used when sorting on the property
- suggest_@{http://www.alfresco.org/model/content/1.0}name = used in query term suggestion

Let's look at a non-dynamic field that is stored, such as the unique ID for the document:



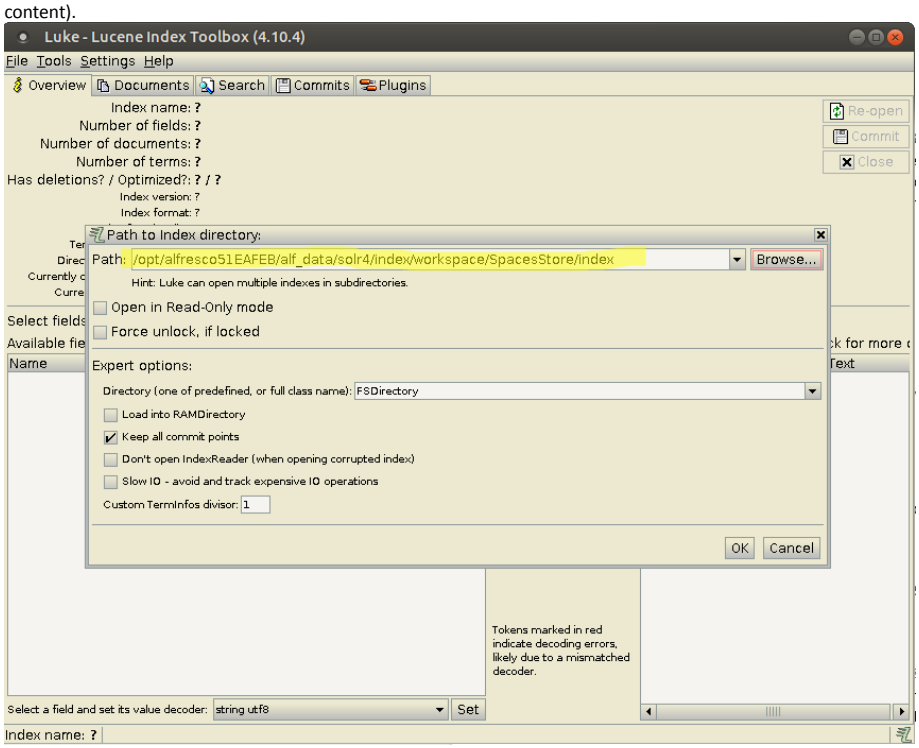
Clicking the **Load Term Info** button gives sample stored values.

Analysing the Lucene Index in more detail

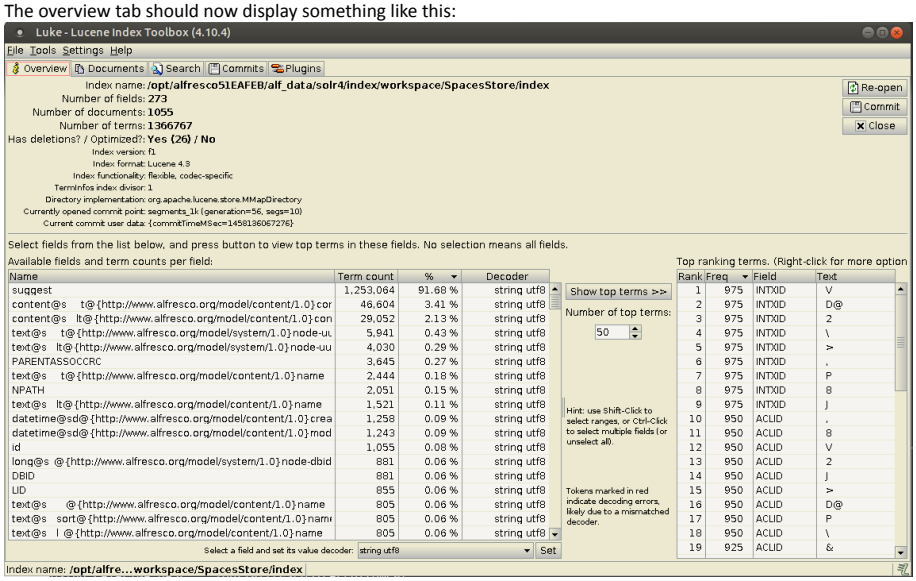
Sometimes it is useful to be able to open up the index files and have a look at the fields that have been indexed etc. This can easily be done with the Luke toolbox. Download the Luke ZIP bundle (**luke-with-deps.tar)** for version 4 of Lucene, such as from the luke-4.10.4 release:
https://github.com/DmitryKey/luke/releases
And extract it in a folder. Then start it up as follows:

```
martin@gravitonian:~/apps/luke$ ls -l
total 8
-rwxrwxr-x 1 martin martin  529 Mar 16  2015 luke.sh
drwxrwxr-x 2 martin martin 4096 Mar 16 14:18 target
martin@gravitonian:~/apps/luke$ ./luke.sh
```

First thing you need to do when Luke starts is to point to where the index files are. Select **<alfrescoinstalldir>/alf_data/solr4/index/workspace/SpacesStore/index** (this is for live content).



Note. the index will not open if you are running Alfresco at the same time!
The overview tab should now display something like this:



In Luke you will be able to see all fields that have been indexed and you can browse through the documents in the index.
This tool is probably only for the very initiated!

Alfresco Solr related directory structure and files

After you have installed Alfresco 5.1 there will be several new directories and configuration files having to do with Solr 4. Let's have a look at them. All directories that we refer to in this section are located under `<alfrescoinstalldir>`, such as for example **/opt/alfresco** or **c:\Alfresco**.

tomcat/webapps/solr4.war (Solr Web App)

This is the customized Alfresco Solr 4 web application that contains tracker components, document content store etc.

tomcat/conf/Catalina/localhost/solr4.xml (Solr Web App Context)

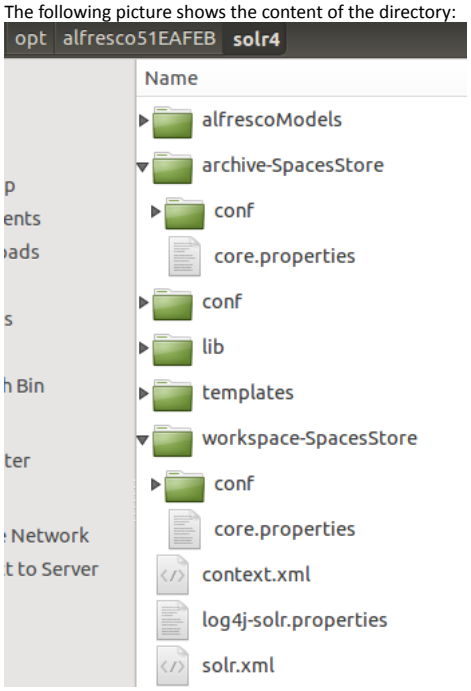
This file defines the Solr 4 web application context and sets up important environment variables. The contents of this file looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<Context debug="0" crossContext="true">
   <Environment name="solr/home" type="java.lang.String" value="/opt/alfresco/solr4" override="true"/>
   <Environment name="solr/model/dir" type="java.lang.String" value="/opt/alfresco/alf_data/solr4/model" override="true"/>
   <Environment name="solr/content/dir" type="java.lang.String" value="/opt/alfresco/alf_data/solr4/content" override="true"/>
</Context>
```

The `solr/home` variable points to where all the configuration is for the cores that are managed by this Solr installation. The `solr/model/dir` variable points to where Solr holds all the content model files that it downloads from Alfresco. The `solr/content/dir` variable points to the directory where the custom Solr Document Content store is located.

solr4/ (Solr Home Directory)

This directory is the Solr 4 home directory and it mainly contains the configuration for the Solr cores and a file called **solr.xml**. In our case there are 2 cores, one for live/workspace content and one for deleted/archived content. As mentioned before, a Solr core holds one Lucene index and the supporting configuration for that index. Pretty much all interactions with Solr are targeted at a specific core.
The following picture shows the content of the directory:



The sub-directories and files in this directory have the following meaning:

Directory/File	Description	Comments
alfrescoModels	Contains the Alfresco content models, such as the cm, cmis, d, and sys models that are fetched from the repo by trackers.	This is the default directory for models fetched from the repo via the trackers. It will not be used if you specify a new location via solr4.xml WAR context configuration parameter (e.g. `<Environment name="solr/model/dir" type="java.lang.String" value="/var/lib/tomcat7/data/model" override="true"/>`).
archive-SpacesStore	The configuration directory for the archive core. The archive store in Alfresco repository contains soft deleted files.	
archive-SpacesStore/core.properties	Contains key information about the core, such as the name.	Solr4 core discovery looks for this file
conf	Contans the shared.properties.sample file which is used in a sharding setup.	More on this in the "*Searching with Alfresco 5.1 Enterprise*" article
lib	Extra libraries that the Solr web application will load on startup	Empty at the moment
templates	Core templates are used to define the base configuration for a new Solr core with some configuration properties.	More on this in the "*Searching with Alfresco 5.1 Enterprise*" article
workspace-SpacesStore	The configuration directory for the workspace core. The workspace store in Alfresco repository contains live content/files.	
workspace-SpacesStore/core.properties	Contains key information about the core, such as the name.	Solr4 core discovery looks for this file
CreateSSLKeystores.txt	Instructions for Generating Solr SSL keystores	
context.xml	Solr Web Application context template for use when installing Solr in separate tomcat.	We will see an example of how to do this later on in this article.
solr.xml	Configuration file specifying the cores to be used by Solr	

solr4/solr.xml (Configuration of all Solr cores)

If you have worked with Solr 1.4 and Alfresco 4 you know that the **solr.xml** configuration file specifies the cores that will be handled by the Solr server:

```
<?xml version="1.0" encoding="UTF-8" ?>
<solr persistent="true" sharedLib="lib" >
   <cores  adminPath="/admin/cores"
           adminHandler="org.alfresco.solr.AlfrescoCoreAdminHandler">
```

```
        <core name="alfresco" instanceDir="workspace-SpacesStore" />
       <core name="archive"  instanceDir="archive-SpacesStore" />
   </cores>
</solr>
```

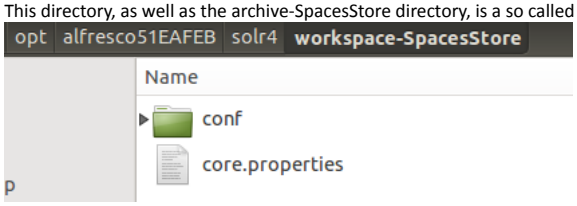
This is not the case anymore with Solr 4, which supports automatic core discovery. The individual `<core>` elements have been replaced with a **core.properties** files in the instance directory (i.e. **archive-SpacesStore** and **workspace-SpacesStore**) for each of your existing Solr cores. These property files can be empty if all defaults are acceptable, in our case they just contain the name of the core. When Solr starts it will look for core.properties files in sub-directories of where the solr.xml file is located.
In an Alfresco 5.1.e installation the **solr.xml** file has the following content:

```
<?xml version='1.0' encoding='UTF-8'?>
<solr>
   <str name="adminHandler">${adminHandler:org.alfresco.solr.AlfrescoCoreAdminHandler}</str>
</solr>
```

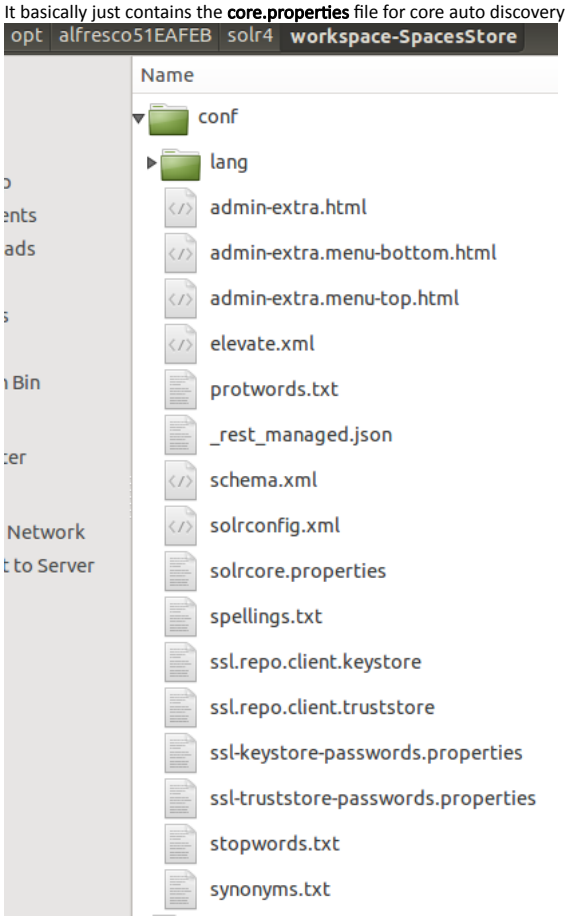
The only thing that is configured here is the custom Admin Handler to support the extra Alfresco Core summary reports in the Core Overview page.
Each instance directory contains the configuration for that Solr core but not the index. The index is located in the **workspace** directory for live content and in the **archive** directory for archived content under **/alf_data/solr4/index**. The index location is configured in the **solrcore.properties** file in each instance directory.

solr4/workspace-SpacesStore (instance directory for a core)

This directory, as well as the archive-SpacesStore directory, is a so called instance directory for a Solr core (i.e. index). For live content it looks like this:



It basically just contains the **core.properties** file for core auto discovery and the **conf** directory with all the core configuration and schema:



The configuration files have the following meaning:

File	Description	Comment
Admin-extra.html, admin-extra.menu-bottom.html, admin-extra.menu-top.html	The content of these files will be statically included into the Admin Extra section for the core Overview page.	The Solr admin page can be accessed via the https://localhost:8443/solr4/#/alfresco URL for the alfresco workspace (live content) core
elevate.xml	This is the configuration file for the `QueryElevationComponent` that enables you to configure the top results for a given query regardless of the normal Lucene scoring. This is sometimes called "sponsored search", "editorial boosting" or "best bets". This component matches the user query text exactly to a configured Map of top results.	If this file is found in the configuration directory, it will only be loaded once at startup. If it is found in Solr's data (core/index) directory, it will be re-loaded at every commit. This search component (i.e. `<searchComponent name="elevator">`) is not in the standard component list for the `/afts` and `/alfresco` request handlers so it must be registered with them in **solrconfig.xml** before you can start using it.
protwords.txt		This filter configuration can be found in the **schema.xml** for the following field type:

	If stemming is used then this file contains protected words that should not be stemmed. The file is loaded and used by the `KeywordMarkerFilterFactory` filter. (Stemming is the process of reducing inflected or sometimes derived words to their stem, base, or root form, for example cats to cat)	`<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">`
_rest_managed.json	This file is used when managed resources are used for a field type in the schema	
schema.xml	This is the Solr schema for the index including field and field type definitions with associated analyzer chains for indexing and searching.	
solrconfig.xml	This is the file that contains most of the parameters for configuring Solr itself. A big part of this file is made up of request handlers, which are defined with `<requestHandler>` elements	
solrcore.properties	Solr core configuration with information about where the Alfresco repository is running, how often it should query the repository, batch sizes etc. Solr supports system property substitution, allowing the launching JVM to specify string substitutions within either of Solr's configuration files. All the properties that need to be substituted can be put into this file.	
spellings.txt	This is the configuration file for the file based spell checker `FileBasedSpellChecker`.	It is not enabled and by default a custom Alfresco spellchecker component is used: `<searchComponent name="spellcheck" class="org.alfresco.solr.component.spellcheck.AlfrescoSpellCheckComponent">`
ssl.repo.client.keystore	This keystore contains the Solr public/private RSA key pair	This is Alfresco <-> Solr integration specific
ssl.repo.client.truststore	This keystore contains the trusted Alfresco Certificate Authority certificate (which has been used to sign both the repository and Solr certificates)	This is Alfresco <-> Solr integration specific
ssl-keystore-passwords.properties	Password information	This is Alfresco <-> Solr integration specific
ssl-truststore-passwords.properties	Password information	This is Alfresco <-> Solr integration specific

stopwords.txt	This file is not used. Instead language specific files are used by each language specific text field definition. The language specific stopwords files can be found in the **/lang** directory.	For english text fields this configuration looks like this in the **schema.xml** file:
		
		`<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">`
		`<analyzer type="index">`
		`<tokenizer class="solr.StandardTokenizerFactory"/>`
		`<filter class="solr.StopFilterFactory" ignoreCase="true" words="lang/stopwords_en.txt" />`
		`...`

synonyms.txt	This file is used by the `SynonymFilterFactory`, which matches strings of tokens (such as for example i-pod) and replaces them with other strings of tokens (such as for example ipod).
	
	The purpose of synonym processing is easily understood. You search for a word that wasn't in the original document (such as for example i-pod) but is synonymous with a word that is indexed (i.e. ipod), so you want that document to match the query.
		This filter is active at query time, see for example the english text field in **schema.xml**:
		
		`<fieldType name="text_en" class="solr.TextField" positionIncrementGap="100">`
		`...`
		`<analyzer type="query">`
		`<tokenizer class="solr.StandardTokenizerFactory"/>`
		`<filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt" ignoreCase="true" expand="true"/>`

solr4/<core>/conf/solrcore.properties

This is the property configuration file for a core, and it supports property substitution. It is usually expected to be located in the **{solr.home}/conf** directory when using a single core, but with multiple cores, as in our case, there will be one **solrcore.properties** file in each core's configuration directory.
Opening the **solrcore.properties** file in the **solr4/workspace-SpacesStore/conf** directory shows a lot of properties where many of them are new as of Solr 4 and Alfresco 5.

Configuring Alfresco Host location and Core (Index) location

Let's start with the properties that control the location of the index and if tracking should be enabled or not:

Property Name	Description	Default Value
data.dir.root	The path to where the index files are located for all cores.	<alfrescoinstalldir>/alf_data/solr4/index
data.dir.store	Index directory path for a specific Alfresco store. These two first properties are used in **solrconfig.xml** to set index directory: `<dataDir>${data.dir.root}/${data.dir.store}</dataDir>`	workspace/SpacesStore for live content index
enable.alfresco.tracking	Tells Solr if it should index Alfresco content in the associated Alfresco repository store or not.	true

The Alfresco host that Solr is talking to when fetching metadata, acl, and content is controlled by the following properties:

Property Name	Description	Default Value
alfresco.host	The hostname for the Alfresco instance that Solr should talk to and index. This is the host where the Alfresco repository is running (i.e. where alfresco.war is deployed and running). In a default installation Alfresco and Solr runs in the same Tomcat and on the same host so host would be set to localhost.	localhost
alfresco.port	The HTTP port for the Alfresco instance that Solr should talk to and index.	8080
alfresco.port.ssl	The HTTPS port for the Alfresco instance that Solr should talk to and index.	8443
alfresco.baseUrl	The base URL path to use when talking to the Alfresco repository. It is used when constructing URLs, such as for example	/alfresco

	http://localhost:8080/alfresco	
alfresco.cron	This is a cron expression that tells Solr how often it should talk to Alfresco and index new or updated metadata and content. The default value means that Solr will talk to Alfresco every 15 seconds. This can be increased to let's say 30 seconds or more when you are re-indexing. This will allow more time for the indexing threads to perform their actions before they get more work on their queue.	0/15 * * * * ? *
alfresco.stores	This is the Alfresco repository store that this core should index.	workspace://SpacesStore for live content
alfresco.lag	Solr indexes are updated in the background. This is the time (in seconds) this Solr full text index is currently behind the repository updates.	1000
alfresco.hole.retention	Each track will revisit all transactions from the timestamp of the last in the index, less this value, to fill in any transactions that may have been missed.	3600000
alfresco.recordUnindexedNodes	TODO	TODO

Configuring Alfresco <-> Solr communication

Communications between the Alfresco Repository and Solr are protected by SSL with mutual authentication. Both the repository and Solr have their own public/private RSA key pair, signed by an Alfresco Certificate Authority, and stored in their own respective key stores. These key stores are bundled with Alfresco. You can also create your own key stores. It is assumed that the keystore files are stored in **alfresco/alf_data/keystore** directory.

The following properties are only relevant when Solr talks to Alfresco over a secure connection:

Property Name	Description	Default Value
alfresco.secureComms	Tells Solr if it should talk to Alfresco over HTTP or HTTPS. Set to `none` if a plain HTTP connection should be used.	https
alfresco.encryption.ssl.keystore.type	This type specifies the CLIENT keystore. Valid types can be those returned by the `java.security.Security.getAlgorithms("KeyStore")` attribute. These types include the following keystore types, and availability depends on the process and platform `java.security` configuration: ● JKS ● JCEKS ● PKCS12 ● PKCS11 (Java crypto device) ● CMSKS ● IBMi5OSKeyStore ● JCERACFKS ● JCECCAKS keystores	JCEKS
alfresco.encryption.ssl.keystore.provider	The Java provider that implements the `type` attribute (for example, JCEKS type). The provider can be left unspecified and the first provider that implements the keystore type specified is used.	
alfresco.encryption.ssl.keystore.location	The CLIENT keystore location reference. If the keystore is file-based, the location can reference any path in the file system of the node where the keystore is located.	ssl.repo.client.keystore
alfresco.encryption.ssl.keystore.passwordFileLocation	The location of the file containing the password that is used to access the CLIENT keystore, also the default that is used to store keys within the keystore.	ssl-keystore-passwords.properties
alfresco.encryption.ssl.truststore.type	See above, but for TRUSTED certificates	JCEKS
alfresco.encryption.ssl.truststore.provider	See above, but for TRUSTED certificates	
alfresco.encryption.ssl.truststore.location	See above, but for TRUSTED certificates	ssl.repo.client.truststore
alfresco.encryption.ssl.truststore.passwordFileLocation	See above, but for TRUSTED certificates	ssl-truststore-passwords.properties

These are pretty much the properties that were available with Alfresco Solr 1.4. With Solr 4 we got loads of new properties.
Here are the properties that control the HTTP connection pool and the thread pool used by the tracking components in Solr:

Property Name	Description	Default Value
alfresco.corePoolSize	Specifies the pool size for multi-threaded tracking and is the number of threads to keep in the pool, even if they are idle. The pool size for the archive core would be set much smaller as there is significantly less to do with soft deleted content than with live content.	workspace = 8 archive = 1
alfresco.maximumPoolSize	Specifies the maximum pool size for multi-threaded tracking.	-1 = not set = will be set the same as alfresco.corePoolSize
alfresco.keepAliveTime	When the number of threads is greater than the core pool size, this is the maximum time (in seconds) that excess idle threads will wait for new tasks before terminating.	120 seconds
alfresco.threadPriority	Specifies the priority that all threads must have on the scale of 1 to 10, where 1 has the lowest priority and 10 has the highest priority.	5
alfresco.threadDaemon	Mark threads as either a daemon threads or user threads. The Java Virtual Machine exits when the only threads running are all daemon threads. If set to false, shutdown is blocked else it is left unblocked.	true
alfresco.workQueueSize	Specifies the maximum number of queued work instances to keep before blocking against further adds.	-1 = unlimited queue
alfresco.maxTotalConnections	Maximum number of HTTP connections that the HTTPClient (org.apache.commons.httpclient) library should use to connect to Alfresco.	200
alfresco.maxHostConnections	Sets the default maximum number of connections allowed for a given host config. (org.apache.commons.httpclient)	200
alfresco.socketTimeout	This property specifies the amount of time Solr tracker will take to notice if the Alfresco web app shuts down first, if Alfresco and Solr are running on the same web application. Sets the default socket timeout in milliseconds which is the timeout for waiting for data. A timeout value of zero is interpreted as an infinite timeout. (org.apache.commons.httpclient)	360000 millisec

Configuring Solr Trackers (Metadata, ACL, and Content)

The tracking of Alfresco events, such as uploading a document, is done in batches and actually consists of a number of steps. Several trackers run and pull information from Alfresco independently. All cores build their index independently from each other.
One property is common for both the Metadata tracker and the ACL tracker (TODO - why not for Content Tracker?):

Property Name	Description	Default Value
alfresco.batch.count	This indicates how many updates should be made to this Solr core (i.e. index) before a commit is executed. This is the main property for controlling Solr communication and how much should be updated in one batch. Each tracking component gets initialized with this batch count.	1000

Metadata tracking (Multi-threaded) to enable metadata searches:

1. Alfresco Repo Call (/api/solr/transactions): **Fetch new transactions** that has happened since last check (reading `alf_transaction` table plus `alf_node` to get count of updated and deleted nodes in a transaction)
2. Calculation: Find out the **total** number of **document/node updates and deletes** for all the transactions
3. Alfresco Repo Call (/api/solr/nodes): Fetch the involved **nodes** and their properties (metadata in aspects and types) so they can be indexed (reading `alf_transaction` table plus `alf_node` and join in `alf_node_properties` to get all the metadata)
4. Async work: Add indexing jobs to the thread pool workers
5. Solr Call: A thread will update the core/index and commit after last update in batch

Here are the properties that can be used to control the metadata tracking process, note that the configuration differs quite a bit between the cores:

Property Name	Description	Default Value
alfresco.transactionDocsBatchSize	Controls how many updated and deleted nodes (repository documents) that should be indexed in one batch. This batch size is checked against all transactions returned in a fetch, so if it gave 200 transactions involving 2500 deletes and/or updates of documents (i.e. nodes). Then the indexing will be done in 5 batches for the workspace core with default value.	workspace = 500 archive = 100
alfresco.nodeBatchSize	When indexing starts, and node properties are fetched (i.e. metadata to be indexed), then this property can be used to control the number of indexing jobs (i.e. delete or update of index) that should be handled by a thread pool worker at a time.	workspace = 100 archive = 10

There are also some other properties that are related to the Metadata tracking and what metadata that should be pulled from the Alfresco Repository and indexed (**note**. this can also be controlled with the `IndexControl` aspect):

Property Name	Description	Default Value
alfresco.metadata.skipDescendantDocsForSpecificTypes	If this property is set to true then all content types specified with the **alfresco.metadata.ignore.datatype.*** pattern will be skipped during indexing. So this is basically an easy way to exclude certain types, and all their properties, from being indexed.	false
alfresco.metadata.ignore.datatype.0	Content type **one** to skip from being indexed	cm:person
alfresco.metadata.ignore.datatype.1	Content type **two** to skip from being indexed	app:configurations
alfresco.metadata.skipDescendantDocsForSpecificAspects	If this property is set to true then all content aspects specified with the **alfresco.metadata.ignore.aspect.*** pattern will be skipped during indexing. So this is basically an easy way to exclude certain aspects, and all their properties, from being indexed.	false
alfresco.metadata.ignore.aspect.0	Content aspect **one** to skip from being indexed	

ACL (Access Control List) tracking (Multi-threaded) to enable query time permission checks:

1. Alfresco Repo Call (/api/solr/aclchangesets): **Fetch new acl changesets** that has happened since last check (reading `alf_acl_change_set` table joining `alf_access_control_list` to get a list of ACL changes)
2. Calculation: Find out the **total** number of **access control lists (ACLs)** for all the changesets
3. Alfresco Repo Call (/api/solr/acls): Fetch the involved ACLs and their properties (id, inherited, etc) so they can be indexed (reading `alf_access_control_list` table)
4. Async work: Add indexing jobs to the thread pool workers
5. Solr Call: A thread will update the core/index and commit after last update in batch

Property Name	Description	Default Value
alfresco.changeSetAclsBatchSize	Controls how many access control lists (ACLs) that should be indexed in one batch. An ACL change set contains one or more ACLs. This batch size is checked against all change sets returned in a fetch, so if it gave 200 changesets involving 2500 ACLs. Then the indexing will be done in 5 batches for the workspace core with default value.	workspace = 500 archive = 100
alfresco.aclBatchSize	When indexing starts, and ACLs are fetched (i.e. authority, node, permission etc), then this property can be used to control the number of indexing jobs (i.e. delete or update of index) that should be handled by a thread pool worker at a time.	workspace = 100 archive = 10

When the Metadata Tracker runs and updates the metadata index it also marks FTS content as out of date or new. This triggers the Content Tracker (also, remember that content tracking is dependent on transformation time - e.g. PDF -> TXT transform time). The Alfresco Solr integrations keeps a cache of the content store. Meaning it keeps a compressed version of the text for each document. This cached content store is used by the Solr Content tracker, which means that it does not always need to talk to the Alfresco Repository. It is also used for example when doing a re-indexing procedure.
Content tracking (Multi-threaded) to enable Full Text Search (FTS):

1. Solr Call: Get Solr documents with unclean content (i.e. Metadata tracker has marked the content as dirty or new)
2. Async work: Add indexing jobs to the thread pool workers
    a. Alfresco Repo Call (/api/solr/textContent): Calls Alfresco to get the text content (if FTS status is *new* then always call Alfresco, if it is *dirty* and updated then check if content ID is different, if it is then call Alfresco, otherwise don't do anything)
    b. Store Content in Solr Content Store: Stores in Solr Caching Content Store (i.e. <alfrescoinstalldir>/alf_data/solr4/content/...)
    c. Mark FTS Status as Clean: Content is now up to date so mark it as clean so it is not processed again by the Content Tracker
3. Solr Call: Commit index updates after last update in batch

Property Name	Description	Default Value
alfresco.contentReadBatchSize	This property controls how many "unclean" Solr documents we want to read in one go	4000
alfresco.contentUpdateBatchSize	This property controls how many content index updates to do in one batch before a commit.	1000

Configuring out-of-the-box Solr in-memory caches

Search your Solr index for “building projects” and you might find it takes hundreds of milliseconds to get results back to you. Try the same search again and get the same results in only a few milliseconds. What you're seeing here is caching: both at the operating system level (as the operating system caches the blocks of the Solr index that you just hit), and also within Solr itself.
There are two cache implementations available for Solr, `LRUCache`, based on a synchronized `LinkedHashMap`, and `FastLRUCache`, based on a `ConcurrentHashMap`. `FastLRUCache` has faster gets and slower puts in single threaded operation and thus is generally faster than `LRUCache` when the hit ratio of the cache is high (> 75%), and may be faster under other scenarios on multi-cpu systems.
Caches are used by a SolrIndexSearcher. When a new searcher is opened, its caches may be pre-populated or "autowarmed" using data from caches in the old searcher. The autowarmCount is the number of items to prepopulate. For the `LRUCache`, the auto warmed items will be the most recently accessed items.
The default configuration for these caches aren’t going to be a silver bullet for solving all our performance problems, so it's important to understand what they do, and to tune them to suit our level of search activity.
The filter query cache (FastLRUCache):
The filter caching features in Solr allow for precise control over how filter queries are handled in order to maximize performance. Solr has the ability to specify if a filter is cached, specify the order filters are evaluated, and specify post filtering.
Adding a filter expressed as a query to a Solr request means adding an additional `fq` parameter for each filter query. For example:

```
http://localhost:8983/solr/demo/select?
  q=cars
  &fq=color:white
  &fq=model:volvo
  &fq=year:[2012 TO *]
```

By default, Solr resolves all of the filters before the main query. Each filter query is looked up individually in Solr’s `filterCache`. Caching each filter query separately accelerates Solr’s query throughput by greatly improving cache hit rates since many types of filters tend to be reused across different requests.
Here are the properties that can be used to size the `filterCache`:

Property Name	Description	Default Value
solr.filterCache.size	The maximum number of entries in the filter cache.	256
solr.filterCache.initialSize	The initial number of entries in the filter cache.	128
solr.filterCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	32

You may want to increase the `size` and `initialSize` of the `filterCache` if you have many users, groups, and tenants.

The query result cache (LRUCache):
Stores sets of document IDs in order returned by a query, a sort, and the range of documents requested. If our “building projects” query returns 100 results, a set of 100 document IDs (integers) will be stored in the query result cache for that query string.
Note that the query cache uses the query string, the user’s authorities, the locale, and ordering to determine the cache key. This means that User A will never use a query cached by user B.
Here are the properties that can be used to size the `queryResultCache`:

Property Name	Description	Default Value
solr.queryResultCache.size	The maximum number of entries in the query result cache.	1024
solr.queryResultCache.initialSize	The initial number of search results to cache.	1024
solr.queryResultMaxDocsCached	The maximum number of documents to cache for any entry in the queryResultCache.	2048
solr.queryResultCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	4

The document cache (LRUCache):
Stores the document fields requested when showing query results. When you ask Solr to do a search, you will generally request one or more stored fields to be returned with your results (using the `fl` parameter). Solr caches those document fields away as well, reducing the time required to service subsequent requests for the same document.
Here are the properties that can be used to size the `documentCache`:

Property Name	Description	Default Value
solr.documentCache.size	The maximum number of document objects to cache.	1024
solr.documentCache.initialSize	The initial number of document objects to cache.	1024
solr.documentCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	512

There is a fourth cache, Lucene's internal cache, the **Field Cache**, but you can't control its behavior. it is managed by Lucene and created when it is first used by the Searcher object. When you ask Solr to sort on a field, every indexed value is put into an array in memory in the Field Cache". This is more of a problem with text than other fields. Not only does this use a lot of memory, but the first time this happens, it takes time to bring in all the values from disk.

Configuring user-defined (Alfresco) Solr in-memory caches

The following properties has to do with the Solr caches specific to the Alfresco-Solr integration.
The authority cache (LRUCache): TODO - not sure about this one, need more info and explanation
This cache is used in authority, such as a user or group, filter generation, such as for example:

```
&fq=AUTHORITY:martin
```

Property Name	Description	Default Value
solr.authorityCache.size	The maximum number of document objects to cache.	128
solr.authorityCache.initialSize	The initial number of document objects to cache.	64
solr.authorityCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	4

The path cache (LRUCache):
This cache will cache `PATH` field queries such as for example:

```
&fq=PATH:"/app:company_home/st:sites/*"
```

Property Name	Description	Default Value
solr.pathCache.size	The maximum number of path query results to cache.	256
solr.pathCache.initialSize	The initial number of path query results to cache.	128
solr.pathCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	32

The owner cache (LRUCache): TODO - not sure about this one, need more info and explanation
This cache will cache `OWNER` and `OWNERSET` field queries such as for example:

```
&fq=OWNER:admin
```

Property Name	Description	Default Value
solr.ownerCache.size	The maximum number of owner query results to cache.	128
solr.ownerCache.initialSize	The initial number of owner query results to cache.	64
solr.ownerCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	0

The reader cache (LRUCache): TODO - not sure about this one, need more info and explanation
This cache will cache `READER` and `READERSET` term queries such as for example:
`&fq=READER:admin`

Property Name	Description	Default Value
solr.readerCache.size	The maximum number of reader query results to cache.	128
solr.readerCache.initialSize	The initial number of reader query results to cache.	64
solr.readerCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	0

The denied cache (LRUCache):TODO - not sure about this one, need more info and explanation
This cache will cache `DENIED` field queries such as for example:
`&fq=DENIED:admin`

Property Name	Description	Default Value
solr.deniedCache.size	The maximum number of denied query results to cache.	128
solr.deniedCache.initialSize	The initial number of denied query results to cache.	64
solr.deniedCache.autowarmCount	The number of items to prepopulate the cache with from previous searches.	0

After tuning the caches we might want to verify that they perform as expected, this section covers that.
If the solution is not going to use Full Text Search then the following property can be used to turn that off:

Property Name	Description	Default Value
alfresco.index.transformContent	If present and set to false the index tracker will not transform any content - only metadata will be indexed (Alfresco 4.1.3 and later)	true

The following property controls the max size of each document text that will be indexed:

Property Name	Description	Default Value
alfresco.contentStreamLimit	Controls the max size of document text that will be indexed. So if your installation is managing very large documents, then this property value might have to be increased to get the full text of the documents indexed.	10 000 000 (10 MB)

solr4/<core instanceDir>/conf/solrconfig.xml

This is the file that contains most of the parameters for configuring Solr itself. A big part of this file is made up of request handlers, which are defined with `<requestHandler` elements. This is also the file where Solr search components are configured and managed. We will look more at how to configure stuff in the requestHandler sections in the "*Searching with Alfresco 5.1 Enterprise*" article.
The first stuff of interest is probably the configuration of the data directory (i.e. Lucene index directory) location for the core:
`<dataDir>${data.dir.root}/${data.dir.store}</dataDir>`
Next section to look at is the general index configuration section `<indexConfig>`.
The first property of interest here is the `<mergeFactor>`, which controls how many segments Lucene should build before merging them together on disk. Default merge factor is 10.
Here is some pros and cons when setting a high and low merge factor:
High value merge factor (e.g. 25):

- **Pros**: Generally improves indexing speed, might also improve replication speed to slaves
- **Cons**: Less frequent merges, resulting in a collection with more index files which may slow searching

Low value merge factor (e.g. 2):

- **Pros**: Smaller number of index files, which speeds up searching.
- **Cons**: More segment merges slow down indexing. It will also be very slow in replication because merged files are copied over again and again, causing high I/O on your slaves.

Next setting that can be useful is the `<ramBufferSizeMB>`, which sets the amount of RAM that may be used by Lucene indexing for buffering added documents and deletions in memory before they are flushed to the disk. The indexing process starts by the creation of an in-memory index in a RAM buffer. When the buffer is full it writes the index as a new segment on disk. The default size of the RAM buffer is set to 100MB (100MB per segment * 10 mergeFactor = 1000MB). Increasing the default ramBufferSizeMB may help indexing performance as there will be less segment merging on disk going on, avoiding intensive I/O.
Another parameter that also plays into this is the `<maxBufferedDocs>` parameter that sets a limit on the number of documents buffered in memory before flushing to disk. So it is worth having a look the type of files uploaded to Alfresco and the size, if we have really small files then we might want to increase the default maxBufferedDocs of 1000 so we don't flush too often, as Lucene will flush based on whichever limit is hit first, the maxBufferedDocs or the ramBufferSizeMB.
There are several Alfresco specific request handlers such as the one that handles Alfresco FTS queries:

```
<requestHandler name="/afts" class="org.apache.solr.handler.component.AlfrescoSearchHandler" lazy="true" >
 <lst name="defaults">
  <str name="defType">afts</str>
  <str name="spellcheck">false</str>
  <str name="spellcheck.extendedResults">false</str>
  <str name="spellcheck.count">5</str>
  <str name="spellcheck.alternativeTermCount">2</str>
  <str name="spellcheck.maxResultsForSuggest">5</str>
  <str name="spellcheck.collate">true</str>
  <str name="spellcheck.collateExtendedResults">true</str>
  <str name="spellcheck.maxCollationTries">5</str>
  <str name="spellcheck.maxCollations">3</str>
  <str name="carrot.title">mltext@m___t@{http://www.alfresco.org/model/content/1.0}title</str>
  <str name="carrot.url">id</str>
  <str name="carrot.snippet">content@s___t@{http://www.alfresco.org/model/content/1.0}content</str>
  <bool name="carrot.produceSummary">true</bool>
  <bool name="carrot.outputSubClusters">false</bool>
 </lst>
 <arr name="components">
   <str>setLocale</str>
   <str>ensureModels</str>
   <str>rewriteFacetParameters</str>
   <str>query</str>
   <str>facet</str>
   <str>mlt</str>
   <str>highlight</str>
   <str>stats</str>
```

```
    <str>debug</str>
    <str>clearLocale</str>
    <str>rewriteFacetCounts</str>
    <!-- <str>addTxId</str> -->
   <str>spellcheck</str>
    <str>setProcessedDenies</str>
    <str>clustering</str>
  </arr>
    <shardHandlerFactory class="org.apache.solr.handler.component.AlfrescoHttpShardHandlerFactory" />
</requestHandler>
```

The query parser for this request handler is specified with the **defType** attribute, which is set to **afts** in the above request handler. You will find the query parsers defined further down in the file, the **afts** one looks like this:

```
<queryParser name="afts" class="org.alfresco.solr.query.AlfrescoFTSQParserPlugin"/>
```

Among the request handlers you will also start to see different search components defined, such as the spell checker component:

```
<searchComponent name="spellcheck" class="org.alfresco.solr.component.spellcheck.AlfrescoSpellCheckComponent">
```

Search components enable a request handler to chain together reusable pieces of functionality to create custom search handlers without writing code. Search components can be reused by multiple instances of request handlers, either by pre-pending (`"first-components"`), appending (`"last-components"`), or replacing (`"components"`) the default list.

solr4/<core instanceDir>/conf/schema.xml

A Solr schema defines the relationship between the fields in a document and a Solr core (i.e. index). The schema identifies the document fields to index in Solr and maps fields to field types. The schema for a core is defined in a **schema.xml** file.
The Alfresco Solr schema have dynamic field definitions to cover all the content model property types:

```
<dynamicField name="any@s_@*"           type="oldStandardAnalysis"
<dynamicField name="encrypted@s_@*"     type="oldStandardAnalysis"
<dynamicField name="int@s_@*"           type="int"
<dynamicField name="long@s_@*"          type="long"
<dynamicField name="float@s_@*"         type="float"
<dynamicField name="double@s_@*"        type="double"
<dynamicField name="date@s_@*"          type="date"
<dynamicField name="datetime@s_@*"      type="date"
<dynamicField name="boolean@s_@*"       type="oldStandardAnalysis"
<dynamicField name="qname@s_@*"         type="oldStandardAnalysis"
<dynamicField name="category@s_@*"      type="oldStandardAnalysis"
<dynamicField name="noderef@s_@*"       type="oldStandardAnalysis"
<dynamicField name="childassocref@s_@*" type="oldStandardAnalysis"
<dynamicField name="assocref@s_@*"      type="oldStandardAnalysis"
<dynamicField name="path@s_@*"          type="oldStandardAnalysis"
<dynamicField name="locale@s_@*"        type="lowercase_id"
<dynamicField name="period@s_@*"        type="oldStandardAnalysis"
```

There are also field definitions for things like:

```
<field name="PARENT"                type="identifier"
<field name="PATH"                  type="path"
<field name="ANCESTOR"              type="identifier"
<field name="QNAME"                 type="path"
<field name="PRIMARYASSOCQNAME"     type="path"
<field name="PRIMARYASSOCTYPEQNAME" type="path"
<field name="ISNODE"                type="identifier"
<field name="ASSOCTYPEQNAME"        type="path"
<field name="PRIMARYPARENT"         type="identifier"
<field name="TYPE"                  type="identifier"
<field name="ASPECT"                type="identifier"
<field name="PROPERTIES"            type="identifier"
<field name="NULLPROPERTIES"        type="identifier"
```

So it is unlikely that you will have to customize the schema very much, if at all. Maybe to make some fields stored, such as `EXCEPTIONMESSAGE` and `EXCEPTIONSTACK`.
For more information about Solr schemas see this section and for more more information about the Alfresco Solr schema see this section.

Logging

This section covers logging of searching and indexing.

Logging search requests

If you want to have a look at the queries that Alfresco is running against Solr when you execute searches, or just click around, in Alfresco Share then enable debug logging as follows in **log4j.properties** (located in <alfrescoinstalldir>/tomcat/webapps/alfresco/WEB-INF/classes):

```
log4j.logger.org.alfresco.repo.search.impl.solr.SolrQueryHTTPClient=debug
```

A log for a full text search on “installation” looks like this:

```
2016-03-16 13:08:12,524  DEBUG [impl.solr.SolrQueryHTTPClient] [http-apr-8080-exec-2] Sent :/solr4/alfresco/afts?
wt=json&fl=DBID%2Cscore&rows=25&df=keywords&start=0&locale=en_GB&alternativeDic=DEFAULT_DICTIONARY&fq=%7B%21afts%7DAUTHORITY_FILTER_FROM_JSON&fq=%7B%21afts%7I
1DAY+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dcreated%3A%5BNOW%2FDAY-
7DAYS+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dcreated%3A%5BNOW%2FDAY-
1MONTH+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dcreated%3A%5BNOW%2FDAY-
6MONTHS+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dcreated%3A%5BNOW%2FDAY-
1YEAR+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dcontent.size%3A%5B0+TO+10240%5D&facet.quer
1DAY+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dmodified%3A%5BNOW%2FDAY-
7DAYS+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dmodified%3A%5BNOW%2FDAY-
1MONTH+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dmodified%3A%5BNOW%2FDAY-
6MONTHS+TO+NOW%2FDAY%2B1DAY%5D&facet.query=%7B%21afts%7D%40%7Bhttp%3A%2F%2Fwww.alfresco.org%2Fmodel%2Fcontent%2F1.0%7Dmodified%3A%5BNOW%2FDAY-
1YEAR+TO+NOW%2FDAY%2B1DAY%5D&fq=%7B%21afts%7D%2BTYPE%3A%22cm%3Acontent%22+OR+%2BTYPE%3A%22cm%3Afolder%22&fq=%7B%21afts%7D-TYPE%3A%22cm%3Athumbnail%22+AND+-
TYPE%3A%22cm%3AfailedThumbnail%22+AND+-TYPE%3A%22cm%3Arating%22+AND+-TYPE%3A%22st%3Asite%22+AND+-ASPECT%3A%22st%3AsiteContainer%22+AND+-
ASPECT%3A%22sys%3Ahidden%22+AND+-cm%3Acreator%3Asystem+AND+-QNAME%3Acomment%5C-*&spellcheck.q=installation&spellcheck=true
```

```
 2016-03-16 13:08:12,524  DEBUG [impl.solr.SolrQueryHTTPClient] [http-apr-8080-exec-2]    with: {"tenants":[""],"locales":
["en_GB"],"defaultNamespace":"http://www.alfresco.org/model/content/1.0","textAttributes":
[],"defaultFTSOperator":"AND","defaultFTSFieldOperator":"AND","anyDenyDenies":false,"query":"installation ","templates":[{"template":"%(cm:name cm:title
cm:description ia:whatEvent ia:descriptionEvent lnk:title lnk:description TEXT TAG)","name":"keywords"}],"allAttributes":
[],"queryConsistency":"DEFAULT","authorities":["GROUP_EVERYONE","ROLE_ADMINISTRATOR","ROLE_AUTHENTICATED","admin"]}
 2016-03-16 13:08:12,525  DEBUG [impl.solr.SolrQueryHTTPClient] [http-apr-8080-exec-2] Got: 25 in 871 ms
```

Logging indexing requests

To see what Solr is doing during indexing enable the following logging in **log4j-solr.properties** (located in <alfrescoinstalldir>/solr4):

```
log4j.logger.org.alfresco.solr.tracker.AclTracker=debug
log4j.logger.org.alfresco.solr.tracker.ContentTracker=debug
log4j.logger.org.alfresco.solr.tracker.MetadataTracker=debug
log4j.logger.org.alfresco.solr.tracker.ModelTracker=debug
```

After a restart of Tomcat the following logging should start to appear:

```
2016-03-16 13:37:00,004  INFO  [solr.tracker.ContentTracker] [SolrTrackerScheduler_Worker-12] total number of docs with content updated: 0
 2016-03-16 13:37:00,004  INFO  [solr.tracker.ContentTracker] [SolrTrackerScheduler_Worker-13] total number of docs with content updated: 0
 2016-03-16 13:37:00,042  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] Verified first acl transaction and timestamp in index
 2016-03-16 13:37:00,042  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] Verified last acl transaction timestamp in index less than or
equal to that of repository.
 2016-03-16 13:37:00,054  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] Verified first acl transaction and timestamp in index
 2016-03-16 13:37:00,054  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] Verified last acl transaction timestamp in index less than or
equal to that of repository.
 2016-03-16 13:37:00,063  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] Verified first transaction and timestamp in index
 2016-03-16 13:37:00,064  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] Verified last transaction timestamp in index less than or
equal to that of repository.
 2016-03-16 13:37:00,067  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] Scanning Acl change sets ...
 2016-03-16 13:37:00,067  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] .... from AclChangeSet [id=18, commitTimeMs=1458054230479,
aclCount=1]
 2016-03-16 13:37:00,067  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] .... to AclChangeSet [id=18, commitTimeMs=1458054230479,
aclCount=1]
 2016-03-16 13:37:00,074  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] Scanning Acl change sets ...
 2016-03-16 13:37:00,074  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] .... from AclChangeSet [id=18, commitTimeMs=1458054230479,
aclCount=1]
 2016-03-16 13:37:00,074  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] .... to AclChangeSet [id=18, commitTimeMs=1458054230479,
aclCount=1]
 2016-03-16 13:37:00,076  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] Verified first transaction and timestamp in index
 2016-03-16 13:37:00,076  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] Verified last transaction timestamp in index less than or
equal to that of repository.
 2016-03-16 13:37:00,086  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] Scanning transactions ...
 2016-03-16 13:37:00,086  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] .... from Transaction [id=73, commitTimeMs=1458111327943,
updates=1, deletes=0]
 2016-03-16 13:37:00,086  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] .... to Transaction [id=117, commitTimeMs=1458111509104,
updates=2, deletes=1]
 2016-03-16 13:37:00,093  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] Scanning transactions ...
 2016-03-16 13:37:00,093  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] .... from Transaction [id=117, commitTimeMs=1458111509104,
updates=2, deletes=1]
 2016-03-16 13:37:00,093  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] .... to Transaction [id=118, commitTimeMs=1458111510446,
updates=2, deletes=1]
 2016-03-16 13:37:00,099  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] Scanning transactions ...
 2016-03-16 13:37:00,099  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] .... from Transaction [id=73, commitTimeMs=1458111327943,
updates=1, deletes=0]
 2016-03-16 13:37:00,099  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] .... to Transaction [id=117, commitTimeMs=1458111509104,
updates=2, deletes=1]
 2016-03-16 13:37:00,101  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] Scanning Acl change sets ...
 2016-03-16 13:37:00,101  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] .... none found after lastTxCommitTime 1458054230479
 2016-03-16 13:37:00,101  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-7] total number of acls updated: 0
 2016-03-16 13:37:00,110  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] Scanning transactions ...
 2016-03-16 13:37:00,110  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] .... from Transaction [id=117, commitTimeMs=1458111509104,
updates=2, deletes=1]
 2016-03-16 13:37:00,110  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] .... to Transaction [id=118, commitTimeMs=1458111510446,
updates=2, deletes=1]
 2016-03-16 13:37:00,125  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] Scanning Acl change sets ...
 2016-03-16 13:37:00,125  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] .... none found after lastTxCommitTime 1458054230479
 2016-03-16 13:37:00,125  INFO  [solr.tracker.AclTracker] [SolrTrackerScheduler_Worker-9] total number of acls updated: 0
 2016-03-16 13:37:00,174  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] Scanning transactions ...
 2016-03-16 13:37:00,174  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] .... none found after lastTxCommitTime 1458111510446
 2016-03-16 13:37:00,174  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-11] total number of docs with metadata updated: 0
 2016-03-16 13:37:00,177  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] Scanning transactions ...
 2016-03-16 13:37:00,177  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] .... none found after lastTxCommitTime 1458111510446
 2016-03-16 13:37:00,177  INFO  [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-16] total number of docs with metadata updated: 0
```

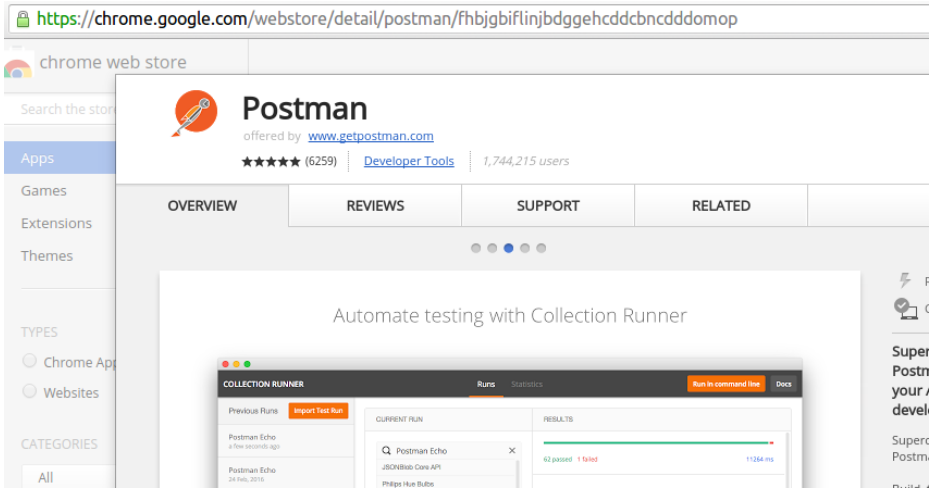
Upload a document to Alfresco and the following type of logs should appear:

```
 2016-03-16 13:47:30,006  INFO  [solr.tracker.ContentTracker] [SolrTrackerScheduler_Worker-28] total number of docs with content updated: 1
 2016-03-16 13:47:30,083  DEBUG [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-31] Node [id=25, nodeRef=workspace://SpacesStore/059ae2fa-6b81-
4517-b75d-9f0a5e7f3422, txnId=124, status=UPDATED, tenant=, aclId=22]
 2016-03-16 13:47:30,083  DEBUG [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-31] Node [id=1045, nodeRef=workspace://SpacesStore/f7564dc3-
2c8f-4272-913b-1595599dddce, txnId=125, status=UPDATED, tenant=, aclId=23]
 2016-03-16 13:47:30,083  DEBUG [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-31] Node [id=1049, nodeRef=workspace://SpacesStore/d7d2c234-
6fbb-4d9c-85a5-b38fbb5b955d, txnId=125, status=UPDATED, tenant=, aclId=23]
 2016-03-16 13:47:30,083  DEBUG [solr.tracker.MetadataTracker] [SolrTrackerScheduler_Worker-31] Node [id=1048, nodeRef=workspace://SpacesStore/ab473fc3-
f90e-44bb-9263-b26ad4da6592, txnId=125, status=DELETED, tenant=, aclId=0]
```

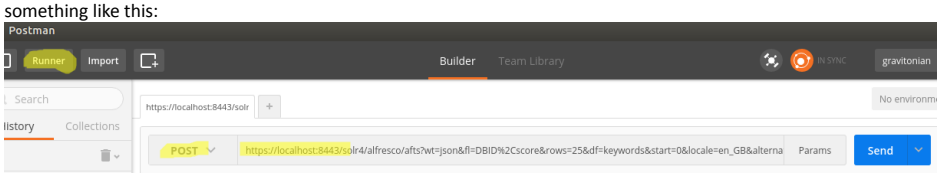
There will be more nodes updated then just the one we upload as the folder we upload to will be updated, and there will be auto-generated nodes such as a thumbnail and a preview.

Debugging Search

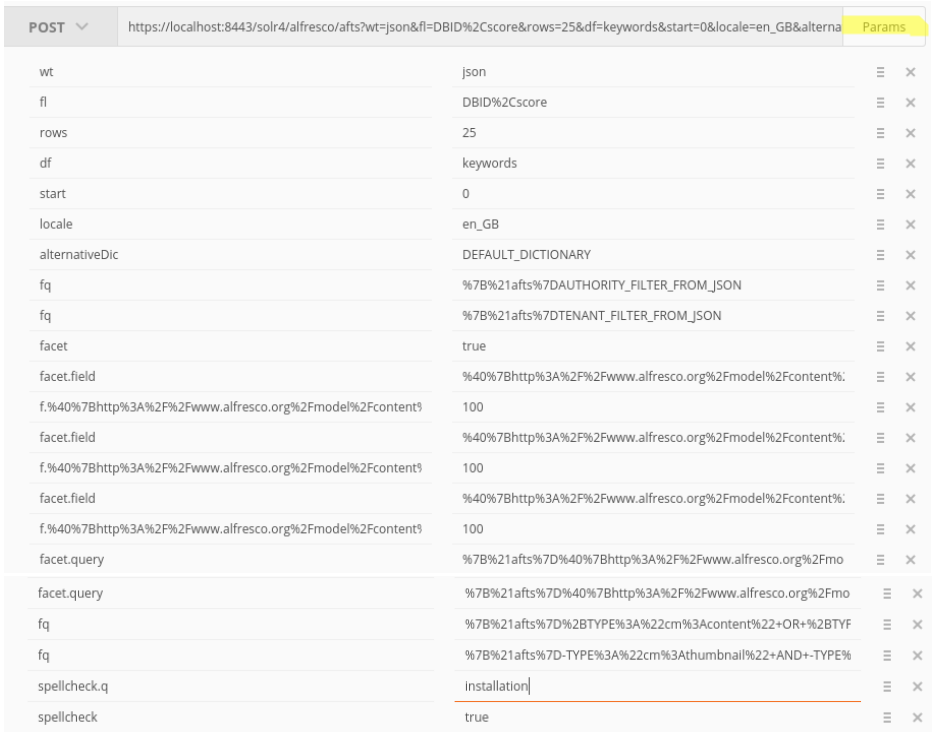
The search queries that are posted to Solr from Alfresco Share can be quite elaborate and difficult to interpret by just looking at the log files. We can use a REST client tool in Chrome called “Postman” to help out with debugging of the search:



After installation (and a free sign-up) we can take the sent part and the with part from the logs (i.e the URL and the POST Payload) and use it to test and interpret the query. Start by clicking the **Runner** button at the top of the Postman tool. Then select POST method. Now copy in the sent bit (`Sent :/solr4/alfresco/afts?wt=json...`) from the logging so you have something like this:



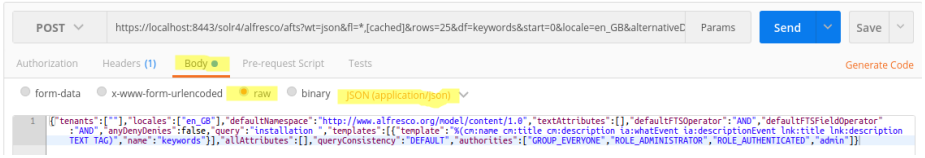
Note that you have to add the https://localhost:8443 before the sent bit. We can now easily get a clearer picture of the URL parameters as we can just click the **Params** button and the Postman tool will display them nicely for us:



Seen like this it is much easier to understand what query parameters that are POSTed to Solr. Here is an explanation of these parameters.

- **wt**: writer type that tells Solr to return JSON as response (could also be **xml**)
- **fl**: field listing that tells Solr what stored fields to return. Remember that not many fields are actually stored (e.g. DBID, score) but you can use a trick to return the complete posted Document. For complete documents in search result use "*,[cached]" but remember that this will return the complete document including indexed text.
- **rows**: maximum number of rows returned in result
- **df**: default schema field to search
- **fq**: filter query, used to filter, here it's used to filter for authorities and tenants, (similar to a where clause in a SQL query)
- **facet**: In order to enable faceting, this parameter must be set to **true**. If this is not done, then the faceting parameters will be ignored.
- **facet.field**: this parameter must be set to a document field's name in order to facet on that field. Repeat this parameter for each field to be faceted on
- **f.@{http://www.alfresco.org/model/content/1.0}modifier.facet.limit**: maximum hits for this facet field
- **spellcheck**: **true** means allow the spellcheck feature
- **spellcheck.q**: the query for the Solr spellcheck component

When the URL is specified and analyzed to have the parameters you want supply the POST payload (i.e the `with: {"tenants":[""],"locales…` part from the logs) by clicking on the **Body** tab and then **raw**. Copy and paste and it should look like this:

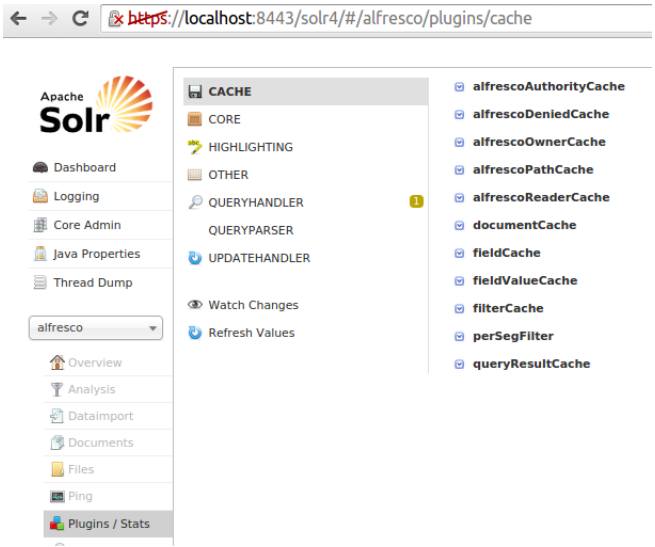


Now we are ready to send the request (note that it will work with HTTPS) by clicking the blue **Send** button. The result will be displayed at the bottom of the screen and you can select that you want it interpreted as JSON:

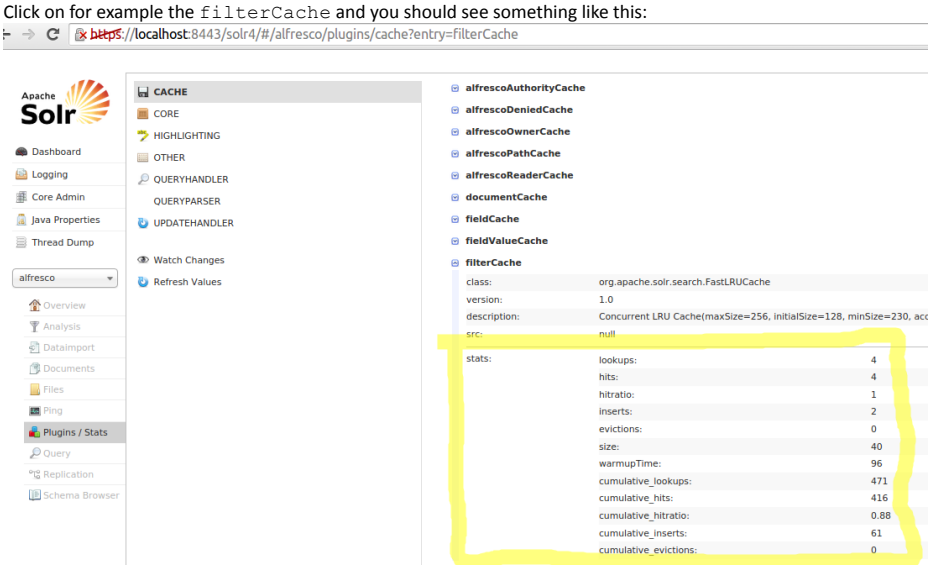
This tool can be quite useful for query testing and analysis.

Measuring cache effectiveness

Before you go live with the system you want to make sure the Solr caches are correctly tuned. This can be done by checking the hit ratio on the caches. The Solr administration webapp has the tool we need for this:



For the **alfresco** core click the **Plugins/Stats** page. Then click the **CACHE** tool. All the Solr caches that is in use by the core will be displayed in a list. The ones with a name starting with **alfresco*** are specific to the Alfresco-Solr integration.
Click on for example the `filterCache` and you should see something like this:



Each set of statistics has a number of different metrics. To determine the effectiveness of a cache, the most interesting figures are:

- The cumulative hit ratio (`cumulative_hitratio`) — The percentage of queries that were satisfied by the cache (a number between 0 and 1, where 1 is ideal). In the above screenshot we can see a hit ratio of 0.88, which is pretty good.
- The cumulative number of inserts (`cumulative_inserts`) — The number of entries added to the cache over its lifetime.

- The cumulative number of evictions (`cumulative_evictions`) — The number of entries removed from the cache over its lifetime. For the filterCache we have not reached max size (256) yet so no evictions.

The ultimate measure of a cache's performance is its hit ratio. You will need to experiment to find your optimal cache sizes, but keep an eye on your hit ratios to make sure you're making things better (not worse). Some tips:

- If you see a high number of evictions relative to inserts, try increasing the size of that cache and monitor the effect on its hit ratio. It might be that entries are being evicted too quickly for your levels of search activity.
- If a cache has a high hit ratio but very few evictions, it might be too large. Try reducing the cache size and see if there's any corresponding change in the hit ratio. In the above screenshot the Alfresco system was just started and after that a few searches was executed. You need to simulate real search usage scenarios before making assumptions on what to do.
- Don't be discouraged if your hit ratio remains low for certain caches. If your queries are generally non-repetitive then no amount of cache sizing is going to get that number up, and you might as well opt for a small cache size.
- Resist the urge to turn the caches off completely, even where the hit rates are low. Some of the caches will still have performance benefits for single queries, so having a small cache set is still worthwhile.
- When changing your cache sizes, try to estimate the worst-case memory usage using the kind of rough calculations found in the Alfresco docs. Make sure that you're not creating a slow memory leak by making your caches too large.

The stats are a good starting point for tuning your caches but you should be aware that by setting the size too large you can see some unwanted Java GC activity. That is why it might be useful to look at the real size of your caches in memory instead of the item count alone.
It's not an exact science, but with a little experimentation and attention to detail you can make big improvements to your overall performance.

Solr memory tuning

Solr uses the Java VM heap for lots of stuff when it is running. Certain things, and configurations, will require a lot of heap memory. Here is a list of stuff that will require more heap memory:

- A large index.
- Frequent updates.
- Very large documents.
- Extensive use of faceting with the default facet.method value.
- Using a lot of different sort parameters.
- Very large Solr caches.
- A large <**ramBufferSizeMB**>.
- Use of Lucene's RAMDirectoryFactory.

Here is a link to documentation covering how to tune the RAM for your installation.

Managing synonym word lists

To support synonym word lists add the synonyms you want to your synonyms.txt file, which is specific for each core (index). Most likely you want to add a synonym word list for the live content index, which is located in **<alfrescoinstalldir>/solr4/workspace-SpacesStore/conf/synonyms.txt**.
Add an entry such as:

```
funny,humorous,comical,hilarious,hysterical
```

Restart the Solr server (i.e. Alfresco in a standard all in one installation) to pick up the new entries.
Now upload a text document with the text containing "funny". You should be able to search on the other synonyms and hit the document.
Note. there is no need to reindex as it is managed during query time. However, there is a potential performance overhead as you are adding extra search terms at query time.

Is there a way to search consistently (transactional)?

As we know, the Solr search system is eventually consistent and up to date with what is happening in the Alfresco repository. So when a search is executed it might not match all newly added files. Is there a way to avoid this problem with eventually consistent index? Yes it is possible to search directly against the database, but only for metadata, not for full text search (FTS).
Alfresco supports the execution of a subset of the CMIS Query Language (CMIS QL) and Alfresco Full Text Search (AFTS) queries directly against the database. This feature is called Transactional Metadata Query (TMQ). Full documentation can be found here.
Out-of-the-box a query will be executed transactional if it is possible. This is controlled by the following two properties that can be custom set in **alfresco-global.properties**:

```
solr.query.fts.queryConsistency=TRANSACTIONAL_IF_POSSIBLE
solr.query.cmis.queryConsistency=TRANSACTIONAL_IF_POSSIBLE
```

Other values are:

- **EVENTUAL** - always use Solr
- **TRANSACTIONAL** - always use the database
- **HYBRID** - search both Solr and Database and merge result

You can track if a query goes to Solr or the Database by turning on logging for the following class in your **log4j.properties**:

```
log4j.logger.org.alfresco.repo.search.impl.solr.DbOrIndexSwitchingQueryLanguage=debug
```

This will log (to the alfresco.log file) how the search system directs queries and falls back from database to index if necessary.

Turning off SSL

A secure connection between Alfresco and Solr might not be needed if everything is running behind a firewall. Turning off SSL might improve performance and simplify the installation.
Turn off SSL in the alfresco webapp (alfresco.war) by opening up the **alfresco-global.properties** file and set the **solr.secureComms** property as follows:

```
### Solr indexing ###
index.subsystem.name=solr4
dir.keystore=${dir.root}/keystore
solr.host=localhost
solr.port.ssl=8443
solr.secureComms=none
```

Then turn off SSL in the Solr webapp (solr4.war) by opening up the <alfrescoinstalldir>/solr4/archive-SpacesStore/conf/**solrcore.properties** file and the <alfrescoinstalldir>/solr4/workspace-SpacesStore/conf/**solrcore.properties** file, then set the following property:
alfresco.secureComms=none

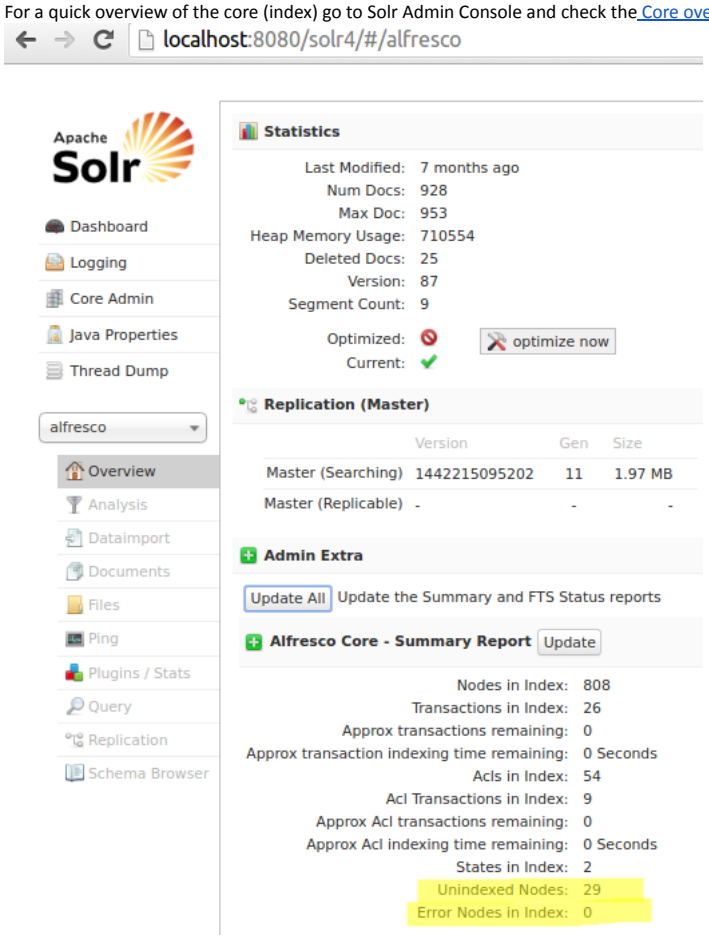
This is all there is too it. It is much easier these days than it used to be. Before you had to also comment out stuff in web.xml for both webapps.
Now restart Alfresco and access the Solr webapp non-secure to verify that it works:
http://localhost:8080/solr4/#/

Troubleshooting indexing problems

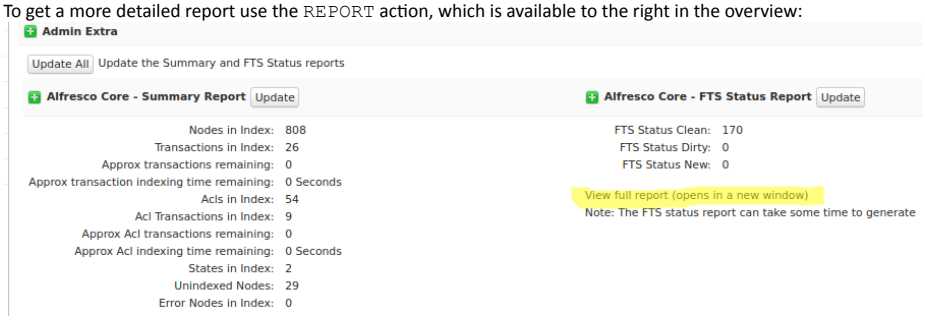
When using Alfresco Solr in production indexing millions of content items it is inevitable that indexing problems will occur. It might be because of corrupt files, insufficient resources, incorrect configuration, etc.

Finding unindexed transactions

For a quick overview of the core (index) go to Solr Admin Console and check the Core overview page:



At the bottom of the Summary report you should immediately be able to see if there are unindexed nodes and if there are errors in the index. In this case there are 29 unindexed nodes in the alfresco core (i.e. live content).
To get a more detailed report use the `REPORT` action, which is available to the right in the overview:



This report can be used to compare the database with the index and generate an overall status report:
http://localhost:8080/solr4/admin/cores?action=REPORT&wt=xml

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">41938</int>
  </lst>
  <lst name="report">
    <lst name="alfresco">
      <str name="Alfresco version">5.1.0 (r@scm-revision@-b@build-number@)</str>
      <long name="DB acl transaction count">9</long>
      <long name="Count of duplicated acl transactions in the index">0</long>
      <long name="Count of acl transactions in the index but not the DB">0</long>
      <long name="Count of missing acl transactions from the Index">0</long>
      <long name="Index acl transaction count">9</long>
      <long name="Index unique acl transaction count">9</long>
      <long name="Last indexed change set commit time">1442214908648</long>
      <str name="Last indexed change set commit date">2015-09-14T08:15:08</str>
      <long name="Last changeset id before holes">-1</long>
      <long name="DB transaction count">26</long>
      <long name="Count of duplicated transactions in the index">0</long>
      <long name="Count of transactions in the index but not the DB">0</long>
      <long name="Count of missing transactions from the Index">0</long>
      <long name="Index transaction count">26</long>
      <long name="Index unique transaction count">26</long>
      <long name="Index node count">808</long>
      <long name="Count of duplicate nodes in the index">0</long>
      <long name="Index error count">0</long>
      <long name="Count of duplicate error docs in the index">0</long>
      <long name="Index unindexed count">29</long>
      <long name="Count of duplicate unindexed docs in the index">0</long>
      <long name="Last indexed transaction commit time">1442215089242</long>
      <str name="Last indexed transaction commit date">2015-09-14T08:18:09</str>
      <long name="Last TX id before holes">-1</long>
      <long name="Node count with FTSStatus Clean">214</long>
      <long name="Node count with FTSStatus Dirty">0</long>
      <long name="Node count with FTSStatus New">0</long>
    </lst>
    <lst name="archive">
```


The result will tell you immediately if you got unindexed transactions and if you got any errors during indexing.
To find out which nodes in particular that are not indexed use the following query:
http://localhost:8080/solr4/alfresco/afts?q=DOC_TYPE:UnindexedNode

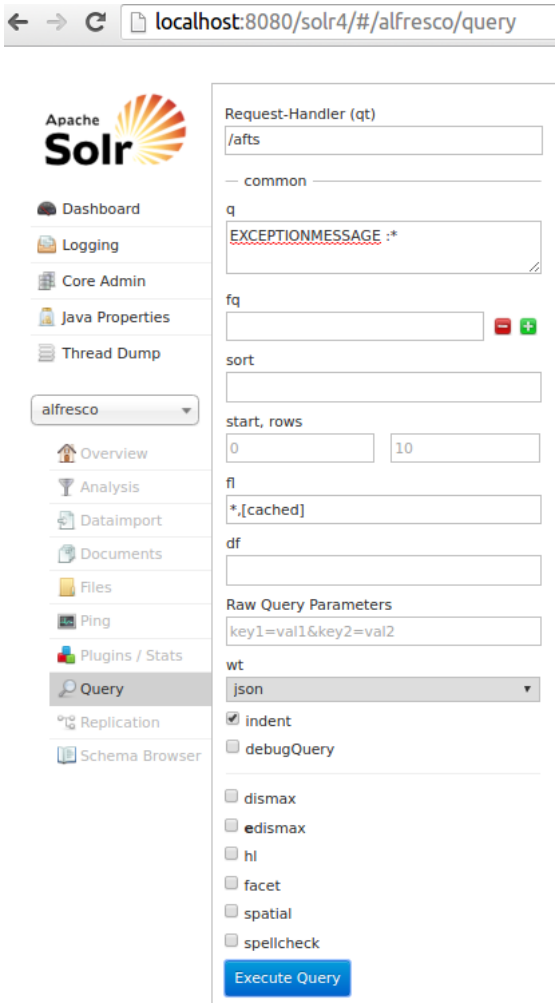

```
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
  </lst>
  <str name="_original_parameters_">
    org.apache.solr.common.params.DefaultSolrParams:
    {params(q=DOC_TYPE:UnindexedNode),defaults(carrot.url=id&spellcheck.cc
  </str>
  <lst name="_field_mappings_"/>
  <lst name="_date_mappings_"/>
  <lst name="_range_mappings_"/>
  <lst name="_pivot_mappings_"/>
  <lst name="_interval_mappings_"/>
  <lst name="_stats_field_mappings_"/>
  <lst name="_stats_facet_mappings_"/>
  <lst name="_facet_function_mappings_"/>
  <result name="response" numFound="29" start="0">
    <doc>
      <str name="id">_DEFAULT_!8000000000000033!80000000000002c1</str>
      <long name="_version_">0</long>
      <long name="DBID">705</long>
    </doc>
    <doc>
      <str name="id">_DEFAULT_!8000000000000033!80000000000002c8</str>
      <long name="_version_">0</long>
      <long name="DBID">712</long>
    </doc>
    <doc>
      <str name="id">_DEFAULT_!8000000000000033!80000000000002c9</str>
      <long name="_version_">0</long>
      <long name="DBID">713</long>
    </doc>
    <doc>
      <str name="id">_DEFAULT_!8000000000000033!80000000000002cb</str>
      <long name="_version_">0</long>
      <long name="DBID">715</long>
    </doc>
    <doc>
      <str name="id">_DEFAULT_!8000000000000033!80000000000002cc</str>
      <long name="_version_">0</long>
      <long name="DBID">716</long>
    </doc>
    <doc>
```


Unindexed nodes are unindexed for a reason, not an error. The nodes will most likely have an aspect and properties that say they are unindexed. For example, some Share site related configuration is unindexed.

Finding out if there are errors

Use the Solr admin pages and schema browser. For example:
http://localhost:8080/solr4/#/alfresco/schema-browser?field=EXCEPTIONMESSAGE

Then click **Load Term Info**. No terms = No problems.
You can also search from the Core Query page:



Note that we specify `fl=*,[cached]` to get the complete document from the cached content store.
Do the same also for:
http://localhost:8080/solr4/#/alfresco/schema-browser?field=EXCEPTIONSTACK

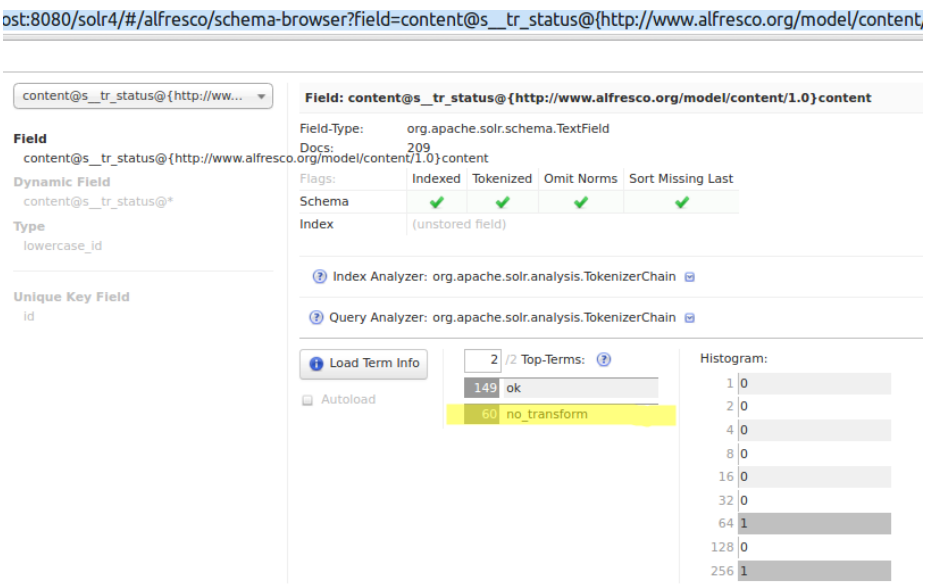
Then click **Load Term Info**. No terms = No problems.

These fields should have been stored (they were in the past and will be in the future). Look in the **schema.xml** for the core, if you see something like this**:**

```
<field name="EXCEPTIONMESSAGE" type="identifier"  indexed="true"  omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="true" />
<field name="EXCEPTIONSTACK" type="identifier" indexed="true"  omitNorms="true"  stored="false"  multiValued="false" sortMissingLast="true" />
```

Then you know these fields are **not** stored and you need to get to the value via the cached content store instead when searching.
If you feel like it you could also update the schema so these fields are stored for the future. Next time you will be able to get the failure related to a particular node via stored fields using a Solr query URL with `fl=*`, and you would then not have to have the whole document returned.
You may find an underlying cause for an indexing error in the Alfresco logs as well as in the index fields. The index fields are more useful if something went wrong on the Solr side.
It is also possible a file has metadata indexed but fails to transform its content into text. This will show up in the Alfresco logs. You can see how content transformation did looking at:
http://localhost:8080/solr4/#/alfresco/schema-browser?field=content@s__tr_status@{http://www.alfresco.org/model/content/1.0}content

Click **Load Term Info**, you should mostly see "ok" and "no_transform":



To find out exactly what nodes experienced errors during indexing use the following query:
http://localhost:8080/solr4/alfresco/afts?q=DOC_TYPE:ErrorNode

Repairing the index

To repair an unindexed or failed transaction use the `FIX` action, which compares the database with the index and identifies any missing or duplicate transactions. It then updates the index by either adding or removing transactions:
http://localhost:8080/solr4/admin/cores?action=FIX
You can also try and fix the nodes with the `RETRY` action:
http://localhost:8080/solr4/admin/cores?action=RETRY

If it does not work to repair the index you might have to resort to rebuilding it from scratch.

Rebuilding the index

If you use Alfresco for some time in production, someone will eventually tell you that you have to reindex after making a change. It comes up over and over again, but what does that actually mean?
Most changes to the schema will require a reindex, unless you only change query-time behavior. A very small subset of changes to solrconfig.xml also require a reindex, and for some changes, a reindex is recommended even when it's not required.
Sometimes it might be necessary to rebuild the whole index. This involves deleting all documents before you begin your indexing process, basically deleting the index directory entirely before you restart Solr.
It's reasonable to wonder why deleting the existing index and building it again from scratch is necessary. The reason is this: when you change your schema nothing happens to the existing data in the index. When Solr tries to access the existing data in the index, it uses the schema as a guide to interpreting that data. If the index contains rows that have a field built with for example the `SortableIntField` class and then Solr tries to access that data with a different class (such as `TrieIntField`), there's a good chance that an unrecoverable error will occur.

Reindex by query

If you just want to re-index a node and you know all about it, such as the node id, then you can directly re-index it like this:
https://localhost:8443/solr4/admin/cores?action=REINDEX&nodeid=1039
The nodeid is not the UUID such as:

sys:node-uuid	d:text	a623c89d-b7be-4266-b4d1-e4780eea3e14

But instead the DB id as follows:

sys:node-dbid	d:long	1039

This is feasible for a few nodes from time to time. But usually you want to bulk re-index nodes based on a type, aspect, property etc. This can be done via re-index by query.
So you can also use a Lucene query to determine what nodes get re-indexed. For example, to re-index all folders you would do this:
https://localhost:8443/solr4/admin/cores?action=REINDEX&query=TYPE:%22cm:folder%22
This query approach is very flexible and the online docs contains more examples.

How to rebuild the cores (indexes)

This can be done and we can use the already cached content to speed up the process (remember Solr 4 has its own store of cached content).
Do as follows:

1. **Stop** Tomcat that runs Solr web application
2. **Remove** index data for workspace core (alf_data/solr4$ rm -rf index/workspace/SpacesStore/)
3. **Remove** index data for archive core (alf_data/solr4$ rm -rf index/archive/SpacesStore/)
4. **Remove** cached content model info (alf_data/solr4$ rm -rf model/*)
5. **Restart** Tomcat that runs Solr web application
6. Wait a bit for Solr to start catching up...

Important. Don't make the mistake of deleting alf_data/solr4/content/_DEFAULT_/db as then Solr has to get all the content from Alfresco repo and transform etc again. So you will lose a lot of time.
You can access the Solr Admin interface to get an idea if the index is up-to-date and ready to go: http://localhost:8080/solr4/#/alfresco

Generating a new keystore

Most likely you will need to re-generate the keystore that is used by Solr and Alfresco for secure communication. This can be done with a script called **generate_keystore.sh** that is located in the <alfrescoinstalldir>/alf_data/keystore directory. You will need to update some paths in the script before using it, and make it executable:

```
martin@gravitonian:/opt/alfresco51EAFEB/alf_data/keystore$ gedit generate_keystores.sh
        ALFRESCO_HOME=/opt/alfresco51EAFEB
        SOLR_HOME=$ALFRESCO_HOME/solr4
martin@gravitonian:/opt/alfresco51EAFEB/alf_data/keystore$ chmod +x generate_keystores.sh
```

Then run it:

```
martin@gravitonian:/opt/alfresco51EAFEB/alf_data/keystore$ ./generate_keystores.sh
/opt/alfresco51EAFEB/tomcat/scripts/ctl.sh : tomcat not running
/opt/alfresco51EAFEB/postgresql/scripts/ctl.sh : postgresql not running
Certificate stored in file </home/martin/ssl.repo.crt>
Certificate stored in file </home/martin/ssl.repo.client.crt>
Certificate was added to keystore
Certificate was added to keystore
Certificate was added to keystore
cp: cannot stat '/opt/alfresco51EAFEB/solr4/templates/store/conf/ssl.repo.client.keystore': No such file or directory
cp: cannot stat '/opt/alfresco51EAFEB/solr4/templates/store/conf/ssl.repo.client.truststore': No such file or directory
cp: cannot create regular file '/opt/alfresco51EAFEB/solr4/templates/store/conf/ssl.repo.client.keystore': No such file or directory
cp: cannot create regular file '/opt/alfresco51EAFEB/solr4/templates/store/conf/ssl.repo.client.truststore': No such file or directory
Certificate update complete
Please ensure that you set dir.keystore=/opt/alfresco51EAFEB/alf_data/keystore in alfresco-global.properties
```

Note that there are no core template called store so you will get a couple of No such file or directory errors, but it should work anyway.
If Solr is running in its own Tomcat then you need to copy over the new keystore to that Tomcat installation and restart:

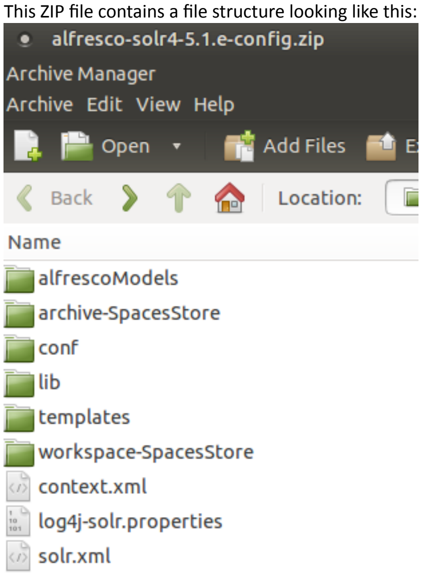
```
martin@gravitonian:/var/lib/tomcat7/data/keystore$ sudo cp /opt/alfresco51EAFEB/alf_data/keystore/* .
martin@gravitonian:/var/lib/tomcat7/data/keystore$ sudo /etc/init.d/tomcat7 restart
 * Stopping Tomcat servlet engine tomcat7                                                                          [ OK ]
 * Starting Tomcat servlet engine tomcat7
[ OK ]
```

Running Solr on a separate Apache Tomcat installation

In a production environment you would want to run Solr in its own Apache Tomcat on its own box. This is good so you can scale and monitor it separately from the application server environment where Alfresco is running, and most of all it allows for easy clustering of the Alfresco application servers as indexes are kept on the Solr box (or by a Solr cluster if you need to scale).
For simplicity, and so you can easily follow along, I will install Solr on a separate Tomcat (8081) on the same box as I run Alfresco Tomcat (8080).
To do this start off by downloading the Solr distribution from the extra Community File list on the Wiki (in case of 5.1.e from https://wiki.alfresco.com/wiki/Community_file_list_201602-GA), it should be two files consisting of the Solr WAR and the Solr configuration:

- alfresco-solr4-5.1.e.war
- alfresco-solr4-5.1.e-config.zip

This ZIP file contains a file structure looking like this:



So basically we got everything that we saw under <alfrescoinstalldir>/solr4 in the full Alfresco installation directory structure, except the index, content, and model directories (i.e. the stuff under <alfrescoinstalldir>/alf_data/solr4) as they are not created until Solr starts and talks to Alfresco about what should be indexed, what content to cache, what custom models exists etc.

Installing Solr 4 on Ubuntu and configure it to talk to Alfresco

The first thing we need to do is download and install Apache Tomcat 7 on the Linux box:

```
$ sudo apt-get install tomcat7
```

This will install Tomcat and its dependencies, such as Java, and it will also create the tomcat7 user. It also starts Tomcat with its default settings.
It will install but not start as I am already running Alfresco Tomcat on port 8080 (default tomcat port). So open up **server.xml** and change port number to 8080 -> **8081** and shutdown port 8005 -> **8006**:

```
/var/lib/tomcat7/conf$ sudo gedit server.xml
```
Note. if you are installing Solr Tomcat on a separate box from Alfresco Tomcat, then you don't need to change the port numbers.
Now start Tomcat 7 as follows:
```
$ sudo /etc/init.d/tomcat7 start
```
If you tail the log the following should print out:
```
martin@gravitonian:/var/lib/tomcat7$ tail -f logs/catalina.out
Apr 08, 2016 5:42:50 AM org.apache.catalina.core.StandardService startInternal
INFO: Starting service Catalina
Apr 08, 2016 5:42:50 AM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.52 (Ubuntu)
Apr 08, 2016 5:42:50 AM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory /var/lib/tomcat7/webapps/ROOT
Apr 08, 2016 5:42:51 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8081"]
Apr 08, 2016 5:42:51 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 938 ms
```

Access http://localhost:8081 and verify that it works. This will displays a minimal "It works" page by default.
Before moving on shutdown Tomcat:
```
$ sudo /etc/init.d/tomcat7 stop
```
Next step is to configure Tomcat for the Apache Solr 4 web application and the Alfresco cores/indexes. I have unpacked the alfresco-solr4-5.1.e-config.zip file in the **~/Downloads/alfresco-solr4-5.1.e-config** directory.
Copy the unpacked files to a directory under **/var/lib/tomcat7** and set them up as accessible by the tomcat7 user:
```
/var/lib/tomcat7$ sudo mkdir data
/var/lib/tomcat7$ sudo chown tomcat7:tomcat7 data/
/var/lib/tomcat7$ cd data/
/var/lib/tomcat7/data$ sudo cp -r ~/Downloads/alfresco-solr4-5.1.e-config/* .
/var/lib/tomcat7/data$ sudo chown -R tomcat7:tomcat7 *
```

We should see something like this now:
```
martin@gravitonian:/var/lib/tomcat7/data$ ls -l
total 36
drwxr-xr-x 2 tomcat7 tomcat7 4096 Apr  8 05:56 alfrescoModels
drwxr-xr-x 3 tomcat7 tomcat7 4096 Apr  8 05:56 archive-SpacesStore
drwxr-xr-x 2 tomcat7 tomcat7 4096 Apr  8 05:56 conf
-rw-r--r-- 1 tomcat7 tomcat7  444 Apr  8 05:56 context.xml
drwxr-xr-x 2 tomcat7 tomcat7 4096 Apr  8 05:56 lib
-rw-r--r-- 1 tomcat7 tomcat7  866 Apr  8 05:56 log4j-solr.properties
-rw-r--r-- 1 tomcat7 tomcat7  147 Apr  8 05:56 solr.xml
drwxr-xr-x 6 tomcat7 tomcat7 4096 Apr  8 05:56 templates
drwxr-xr-x 3 tomcat7 tomcat7 4096 Apr  8 05:56 workspace-SpacesStore
```

Now, setup Tomcat to deploy the Solr web application, the Alfresco Solr Configuration ZIP distribution comes with a web application context file that we can use:
```
/var/lib/tomcat7$ sudo cp data/context.xml conf/Catalina/localhost/solr4.xml
/var/lib/tomcat7$ cd conf/Catalina/localhost/
/var/lib/tomcat7/conf/Catalina/localhost$ sudo chown tomcat7:tomcat7 solr4.xml
```

Update the **solr4.xml** so paths match the installation, set the location of the Solr war file and the location of the Solr home directory:
```
<?xml version="1.0" encoding="utf-8"?>
<Context debug="0" crossContext="true">
  <Environment name="solr/home" type="java.lang.String" value="/var/lib/tomcat7/data" override="true"/>
  <Environment name="solr/model/dir" type="java.lang.String" value="/var/lib/tomcat7/data/model" override="true"/>
  <Environment name="solr/content/dir" type="java.lang.String" value="/var/lib/tomcat7/data/content" override="true"/>
</Context>
```

Next thing we need to do is update each core's configuration and tell it where the index data dir is and where Alfresco is running:

```
/var/lib/tomcat7/data/workspace-SpacesStore/conf$ sudo gedit solrcore.properties
```

Then set the following properties (the IP address is where my Alfresco Tomcat server is running):
```
data.dir.root=/var/lib/tomcat7/data/index
alfresco.host=85.225.85.126
alfresco.secureComms=https
```

Note that the Alfresco Solr 4 Configuration distribution comes with SSL turned off, so we need to turn it on by setting the `alfresco.secureComms` property. Then set these properties to the same values for the archive core:

```
/var/lib/tomcat7/data/archive-SpacesStore/conf$ sudo gedit solrcore.properties
```

Solr is going to write a log and we need to configure it so it is written to a place where it has permission. Open the /var/lib/tomcat7/data/**log4j-solr.properties** file and update the following like:

```
log4j.appender.File.File=/var/lib/tomcat7/logs/solr.log
```

Alfresco Repository <-> Solr communications are protected by SSL with mutual authentication out of the box. Both the repository and Solr have their own private/public RSA key pair, signed by an Alfresco Certificate Authority.

We need to copy the Alfresco Repository keystore to the new Solr Tomcat installation so it can be used to manage HTTPS connections (this keystore is not part of the Alfresco Solr 4 Configuration distribution). It is normally found under <alfrescoinstalldir>/alf_data/keystore or under <alfrescoinstalldir>/tomcat/webapps/alfresco/WEB-INF/classes/alfresco/keystore. I am going to copy it from the **<alfrescoinstalldir>/alf_data/keystore** under the Alfresco 5.1 installation:

```
/var/lib/tomcat7/data$ sudo cp -r /opt/alfresco/alf_data/keystore .
/var/lib/tomcat7/data$ sudo chown -R tomcat7:tomcat7 keystore/
```

So for Alfresco to be able to talk over HTTPS with Solr we need to configure that in **server.xml**:
```
/var/lib/tomcat7/conf$ sudo gedit server.xml
```
Define a new SSL HTTP Connector on port **8444** (or 8443 if installing on separate box) as follows, specifying the location of the the copied keystore files:
```
<Connector port="8444" URIEncoding="UTF-8" protocol="org.apache.coyote.http11.Http11Protocol"
```

```
       SSLEnabled="true"
       maxThreads="150"
       scheme="https"
       keystoreFile="/var/lib/tomcat7/data/keystore/ssl.keystore" keystorePass="kT9X6oe68t" keystoreType="JCEKS"
       secure="true"
       connectionTimeout="240000" truststoreFile="/var/lib/tomcat7/data/keystore/ssl.truststore" truststorePass="kT9X6oe68t" truststoreType="JCEKS"
        clientAuth="want"
       sslProtocol="TLS"
       allowUnsafeLegacyRenegotiation="true"
       maxHttpHeaderSize="32768"
       maxSavePostSize="-1" />
```

Update redirect port for the other plain HTTP connector (if on same box):

```
<Connector port="8081" protocol="HTTP/1.1"
       connectionTimeout="20000"
       URIEncoding="UTF-8"
       redirectPort="8444" />
```

Add the following user to the tomcat-users.xml file located in the /var/lib/tomcat7/conf directory, this will allow the Alfresco Repository to SSL authenticate with Solr:

```
<tomcat-users>
  <user username="CN=Alfresco Repository, OU=Unknown, O=Alfresco Software Ltd., L=Maidenhead, ST=UK, C=GB" roles="repository" password="null"/>
</tomcat-users>
```

This should be all that is needed to setup Alfresco Solr 4 in a separate Tomcat. Make sure Alfresco is running and that https://85.225.85.126:8443/alfresco is reachable (the 85.225.85.126 IP is where Alfresco repo is running), if it is running on a separate box try telnet into the box:

```
$ telnet 85.225.85.126 8443
Trying 85.225.85.126...
```

If it hangs like this then you need to open up the firewall for incoming HTTPS connections.

Now last thing to do before we can start Solr 4 is to copy over the customized Solr 4 WAR from the download dir (**note the name change of the WAR file so it matches the solr4.xml context file created earlier**):

```
/var/lib/tomcat7/webapps$ sudo cp ~/Downloads/alfresco-solr4-5.1.e.war ./solr4.war
/var/lib/tomcat7/webapps$ sudo chown -R tomcat7:tomcat7 solr4.war
```

Then start Solr 4:

```
$ sudo /etc/init.d/tomcat7 start
```

If you see the following in the logs:

```
Exception in thread "SolrTrackingPool-alfresco-MetadataTracker-5" java.lang.OutOfMemoryError: Java heap space
```

Then you need to tweak the Java memory settings for Tomcat, the default 128MB is not going cut it. Set up the following JAVA_OPTS so you get enough memory to run Alfresco Solr:

```
JAVA_OPTS="-Djava.awt.headless=true -Xmx512m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC"
```

In a debian based system like Ubuntu I do this by updating the following file:

```
$ sudo gedit /etc/default/tomcat7
```

Note that it is not the actual boot script but the place where you can specify default values for variables.

Then restart Tomcat.

We should now see the following directories created during the indexing and content caching processes:

```
/var/lib/tomcat7/data/index/workspace
/var/lib/tomcat7/data/index/archive
/var/lib/tomcat7/data/content
/var/lib/tomcat7/data/model
```

Configure Alfresco Server to use a stand-alone Solr server

We now got Solr on a separate Tomcat installation talking to Alfresco and indexing the content store. However, Alfresco is still using the Solr that came with the standard installation so we need to tell it about the new Solr server.
Open up **alfresco-global.properties** located in the <alfrescoinstalldir>/tomcat/shared/classes directory and add the `solr.host` property and `solr.port.ssl` property specifying the new Solr host:

```
### Solr indexing ###
index.subsystem.name=solr
dir.keystore=${dir.root}/keystore
solr.host=85.225.85.126
solr.port.ssl=8444
```

Then disable the Solr web app running locally by renaming **solr4.xml** in <alfrescoinstalldir>/tomcat/conf/Catalina/localhost to **solr4.xml.bak**. And remove the <alfrescoinstalldir>**/tomcat/webapps/solr4** directory and rename solr4.war to **solr4.war.bak**.

Now restart Alfresco and it should start talking to the separate Solr installation when doing searches etc.

How to upgrade Alfresco 4 with Solr 1.4 to Alfresco 5.1 with Solr4

If you already got an older Alfresco 4 installation with Solr 1.4 you might want to upgrade it to newest Alfresco and Solr. Here are the online docs for that.
Note. the indexes are not compatible and a mandatory full reindexing is required.