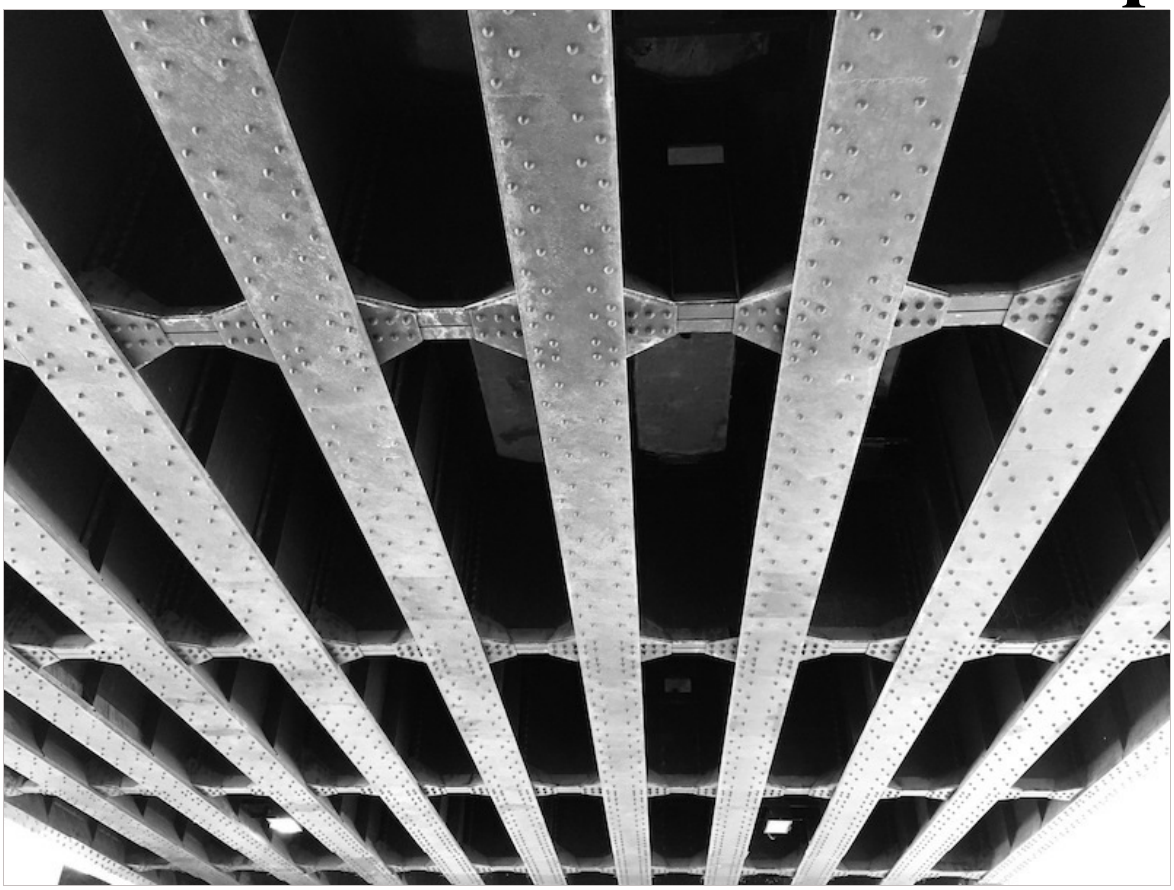
Solr for newbies workshop



Hector Correa
Senior Library Applications Developer
Brown University
hector_correa@brown.edu
http://hectorcorrea.com/solr-for-newbies

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

Table of Contents

- PART I: INTRODUCTION
  - What is Solr
    - Solr's document model
    - Inverted index
    - What is Lucene
  - Installing Solr for the first time
    - Prerequisites
    - Installing Java
    - Installing Solr
    - Let's get Solr started
    - Adding Solr to your path (optional)
  - Creating our first Solr core
  - Adding documents to Solr
  - Searching for documents
    - Fetching data
    - Selecting what fields to fetch
    - Filtering the documents to fetch
    - Getting facets
  - Updating documents
  - Deleting documents
- PART II: SCHEMA
  - Fields in our schema
    - Field: id
    - Field: title_txt_en
  - Analyzers, Tokenizers, and Filters
    - Tokenizers
    - Filters
    - Putting it all together
  - Handling text in Chinese, Japanese, and Korean (optional)
  - Stored vs indexed fields (optional)
  - Customizing our schema
    - Recreating our Solr core
    - Handling `_txts_en` fields
    - Customizing the title field
    - Customizing the author fields
    - Customizing the subject and publisher fields
    - Testing our changes
    - Testing changes to the title field
    - Testing changes to the author field
  - What are others doing
- PART III: SEARCHING
  - Query Parsers
  - Basic searching in Solr
    - q and fq parameters
    - the qf parameter
    - debugQuery
    - Ranking of documents
    - Default Field (optional)
    - Filtering with ranges
    - Minimum match (optional)
    - Where to find more
  - Facets

    - Hit highlighting
- PART IV: MISCELLANEOUS (optional)
    - Solr directories
        - Your bibdata core
    - Synonyms
        - Life without synonyms
        - Adding synonyms
    - Core-specific configuration
        - Request Handlers
        - LocalParams and dereferencing
        - Search Components
    - Spellchecker
    - Sources and where to find more
        - Sample data

PART I: INTRODUCTION

What is Solr

Solr is an open source *search engine* developed by the Apache Software Foundation. On its home page Solr advertises itself as

```
Solr is the popular, blazing-fast,
open source enterprise search platform built on Apache Lucene.
```

and the book Solr in Action describes Solr as

```
Solr is a scalable, ready-to-deploy enterprise search engine
that's optimized to search large volumes of text-centric data
and return results sorted by relevance [p. 4]
```

The fact that Solr is a search engine means that there is a strong focus on speed, large volumes of text data, and the ability to sort the results by relevance.

Although Solr could technically be described as a NoSQL database (i.e. it allows us to store and retrieve data in a non-relational form) it is better to think of it as a search engine to emphasize the fact that it is better suited for text-centric and read-mostly environments [Solr in Action, p. 4].

Solr's document model

Solr uses a document model to represent data. Documents are Solr's basic unit of information and they can contain different fields depending on what information they represent. For example a book in a library catalog stored as a document in Solr might contain fields for author, title, and subjects, whereas information about a house in a real estate system using Solr might include fields for address, taxes, price, and number of rooms.

Something important to know about documents in Solr is that they are self-contained and don't contain nested fields:

```
a document in a search engine like Solr has a flat structure and doesn't
depend on other documents. The flat concept is slightly relaxed in Solr,
in that a field can have multiple values, but fields don't contain
subfields. You can store multiple values in a single field, but you
can't nest fields inside of other fields. [Solr in Action, p. 6]
```

This is different from other document stores, like MongoDB, that allow nested fields inside a document.

Inverted index

Search engines like Solr use a data structure called *inverted index* to support fast retrieval of documents even with complex query expression on large datasets. The basic idea of an inverted index is to use the *terms* inside a document as the *key* of the index rather than the *document's ID* as the key.

Let's illustrate this with an example. Suppose we have three books that we want to index. With a traditional index we would create something like this:

```
ID      TITLE
--      ----------------
1       DC guide for dogs
2       DC tour guide
3       cats and dogs
```

With an inverted index we would take each of the words in the title of our books and use those words as the index key:

```
KEY     DOCUMENT ID
----    -----------------
DC      1, 2
guide   1, 2
dogs    1, 3
tour    2
cats    3
```

Notice that the inverted index allow us to do searches for individual *words within the title*. For example a search for the word "guide" immediately tell us that documents 1 and 2 are a match. Likewise a search for "tour" will tells that document 2 is a match.

Chapter 3 in Solr in Action has a more comprehensive explanation of how Solr uses inverted indexes to allow for partial matches as well as to aid with the ranking of the results.

What is Lucene

The core functionality that Solr makes available is provided by a Java library called Lucene. Lucene is the brains behind the "indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities" that we will see in this tutorial.

But Lucene is a Java Library than can only be used from other Java programs. Solr on the other hand is a wrapper around Lucene that allows us to use the Lucene functionality from any programming language that can submit HTTP requests.

```
                                          --------------------
                                          | Java Runtime     |
[client application] ----> HTTP request ----> | Solr --> Lucene |
                                          --------------------
```

In this diagram the *client application* could be a program written in Ruby or Python. In fact, as we will see throughout this tutorial, it can also be a system utility like cURL or a web browser. Anything that can submit HTTP requests can communicate with Solr.

Installing Solr for the first time

Prerequisites

To run Solr on your machine you need to have the Java Development Kit (JDK) installed. To verify if that the JDK is installed run the following command from the Terminal (aka Command Prompt if you are on Windows):

```
$ java --version

  #
  # java version "11.0.2" 2019-01-15 LTS
  # Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
  # Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)
  #
```

If the JDK is installed on your machine you'll see the text indicating the version that you have (e.g. "11.0.2" above). The specific Java version does not matter much as long as the previous command displays a version of Java, you should be OK.

If you don't have the JDK installed you'll see something like

```
  #
  # -bash: java: command not found
  #
```

Note for Mac users: After running the `java -version` command above if the JDK is not installed the Mac might

give a prompt to install Java. If you follow the prompt's instructions you will be installing the Java Runtime Environment (JRE) which *it is not what you need to run Solr*. It won't hurt to install the JRE but you it won't help you either. You can ignore that prompt.

If Java *is installed* on your machine skip the "Installing Java" section below and jump to the "Installing Solr" section. If Java *is not installed* on your machine follow the steps below to install it.

Installing Java

To install the Java Development Kit (JDK) go to http://www.oracle.com/technetwork/java/javase/downloads/index.html and click the "DOWNLOAD" button at the top of the page to download the "Java Platform (JDK) 11".

From there, under the "Java SE Development Kit 11.0.2", accept the license agreement and select the file appropriated for your operating system. For Mac download the ".dmg" file (`jdk-11.0.2_osx-x64_bin.dmg`) and for Windows download the ".exe" file (`jdk-11.0.2_windows-x64_bin.exe`).

Run the installer that you downloaded and follow the instructions on screen. Once the installer has completed, *open a new Terminal window* and run the `java -version` command again. You should see the text with the Java version number this time.

Installing Solr

You can find download links for Solr at the Apache Solr site. To make it easy, below are the steps to download and install version 7.4 which is the one that we will be using.

First, download Solr and save it to a file.

```
$ cd
$ curl http://archive.apache.org/dist/lucene/solr/7.4.0/solr-7.4.0.zip > solr-7.4.0.zip

  #
  # You'll see something like this...
  # % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
  #                                 Dload  Upload   Total   Spent    Left  Speed
  # 100  146M  100  146M    0     0  7081k      0  0:00:21  0:00:21 --:--:-- 8597k
  #
```

Then unzip the downloaded file with the following command:

```
$ unzip solr-7.4.0.zip

  #
  # A ton of information will be displayed here as Solr is being
  # decompressed/unzipped. Most of the lines will say something like
  # "inflating: solr-7.4.0/the-name-of-a-file"
  #
```

Now that Solr has been installed on your machine you will have a folder named `solr-7.4.0`. This folder has the files to run and configure Solr. The Solr binaries (i.e. the Java JAR files) are under `solr-7.4.0/dist` but for the most part we will use the utilities under `solr-7.4.0/bin` to start and stop Solr.

First, let's make sure we can run Solr by executing the `solr` shell script with the `status` parameter:

```
$ cd ~/solr-7.4.0/bin
$ ./solr status

  #
  # No Solr nodes are running.
  #
```

The "No Solr nodes are running" message is a bit anticlimactic but it's exactly what we want since it indicates

that Solr is ready to be run.

Note for Windows users: In Windows use the `solr.cmd` batch file instead of the `solr` shell script, in other words, use `solr.cmd status` instead of `./solr status`.

Let's get Solr started

To start Solr run the `solr` script again but with the `start` parameter:

```
$ cd ~/solr-7.4.0/bin
$ ./solr start

  #
  # Waiting up to 180 seconds to see Solr running on port 8983 [-]
  # Started Solr server on port 8983 (pid=85830). Happy searching!
  #
```

Notice that the message says that Solr is now running on port `8983`.

You can validate this by opening your browser and going to http://localhost:8983/. This will display the Solr Admin page from where you can perform administrative tasks as well as add, update, and query data from Solr.

You can also issue the `status` command again from the Terminal and Solr will report something like this:

```
$ cd ~/solr-7.4.0/bin
$ ./solr status

  # Found 1 Solr nodes:
  #
  # Solr process 86348 running on port 8983
  # {
  # "solr_home":"/some/path/solr-7.4.0/server/solr",
  # "version":"7.4.0 84..0659 - ubuntu - 2017-10-13 16:15:59",
  # "startTime":"2017-11-11T22:12:15.497Z",
  # "uptime":"0 days, 0 hours, 0 minutes, 12 seconds",
  # "memory":"26.4 MB (%5.4) of 490.7 MB"}
  #
```

Notice how Solr now reports that it has "Found 1 Solr node". Yay!

Adding Solr to your path (optional)

In the previous examples we always made sure we were at the Solr `bin` folder in order to run the Solr commands. You can eliminate this step by making sure Solr is in your PATH. For example if Solr is installed on your home folder (`~/solr-7.4.0`) you can run the following commands:

```
$ cd
$ PATH=~/solr-7.4.0/bin:$PATH
$ which solr

  #
  # /your-home-folder/solr-7.4.0/bin/solr
  #
```

Notice that setting the PATH this way will make it available for your *current* Terminal session. You might want to edit the PATH setting in your `~/.bash_profile` or `~/.bashrc` to make the change permanent.

If you don't do this you will need to make sure that you always refer to Solr with the full path, for example `~/solr-7.4.0/bin/solr`.

Creating our first Solr core

Solr uses the concept of *cores* to represent independent environments in which we configure data schemas and

store data. This is similar to the concept of a "database" in MySQL or PostgreSQL.

For our purposes, let's create a core named `bibdata` as follows (notice these commands require that Solr be running, if you stopped it, make sure you run `solr start` first)

```
$ cd ~/solr-7.4.0/bin
$ ./solr create -c bibdata

  #
  # WARNING: Using _default configset. Data driven schema functionality is enabled by default, which is
  #          NOT RECOMMENDED for production use.
  #
  #          To turn it off:
  #             curl http://localhost:8983/solr/bibdata/config -d '{"set-user-property":
{"update.autoCreateFields":"false"}}'
  #
  # Created new core 'bibdata'
  #
```

Now we have a new core available to store documents. We'll ignore the warning because we are not in production, but we'll discuss this later on.

For now our core is empty (since we haven't added any thing to it) and you can check this with the following command from the terminal:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*:*'

  #
  # {
  #  "responseHeader":{
  #    "status":0,
  #    "QTime":0,
  #    "params":{
  #      "q":"*:*"}},
  #  "response":{"numFound":0,"start":0,"docs":[]
  #  }}
  #
```

(or you can also point your browser to http://localhost:8983/solr#bibdata/query and click the "Execute Query" button at the bottom of the page)

in either case you'll see `"numFound":0` indicating that there are no documents on it.

Adding documents to Solr

Now let's add a few documents to this `bibdata` core. First, download this sample data file:

```
$ cd ~/solr-7.4.0/bin
$ curl 'https://raw.githubusercontent.com/hectorcorrea/solr-for-newbies/master/books.json' > books.json

  #
  # You'll see something like this...
  #   % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
  #                                  Dload  Upload   Total   Spent    Left  Speed
  # 100  1998  100  1998    0     0   5561      0 --:--:-- --:--:-- --:--:--  5581
  #
```

File `books.json` contains a small sample data a set with information about a few thousand books. You can take a look at it via `cat books.json` or using your text editor of choice. Below is an example on one of the books in this file:

```
{
  "id":"00000007",
  "author_txt_en":"Guiney, Louise Imogen,",
  "authorDate_s":"1861-1920.",
```

```
  "title_txt_en":"The martyrs' idyl, and shorter poems,",
  "responsibility_txt_en":"by Louise Imogen Guiney.",
  "publisher_txt_en":"Boston"
}
```

Then, import this file to our `bibdata` core with the `post` utility that Solr provides out of the box (Windows users see note below):

```
$ cd ~/solr-7.4.0/bin
$ ./post -c bibdata books.json

  #
  # (some text here...)
  # POSTing file books.json (application/json) to [base]/json/docs
  # 1 files indexed.
  # COMMITting Solr index changes to http://localhost:8983/solr/bibdata/update...
  # Time spent: 0:00:00.324
  #
```

Now if we run our query again we should see some results

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*:*'

  # Response would be something like...
  # {
  #   "responseHeader":{
  #     "status":0,
  #     "QTime":36,
  #     "params":{
  #       "q":"*:*"}},
  #   "response":{"numFound":1000,"start":0,"docs":[
  #       ... lots of information will display here ...
  #     ]}
  # }
  #
```

Notice how the number of documents found is greater than zero (e.g. `"numFound":1000`)

Note for Windows users: Unfortunately the `post` utility that comes out the box with Solr only works for Linux and Mac. However, there is another `post` utility buried under the `exampledocs` folder in Solr that we can use in Windows. Here is what you'll need to to:

```
> cd C:\Users\you\solr-7.4.0\examples\exampledocs
> copy path\to\books.json .
> java -Dtype=application/json -Dc=bibdata -jar post.jar books.json

  #
  # you should see something along the lines of
  #
  # POSTing file books.json
  # 1 files indexed
  # COMMITting Solr index changes to http://...
  #
```

Searching for documents

Now that we have added a few documents to our `bibdata` core we can query Solr for those documents. In a subsequent section we'll explore more advanced searching options and how our schema definition is key to enable different kind of searches, but for now we'll start with a few basic searches to get familiar with the way querying works in Solr.

If you look at the content of the `books.json` file that we imported into our `bibdata` core you'll notice that the documents have the following fields:

- **id**: string to identify each document (MARC 001)

- **author_txt_en**: string for the main author (MARC 100a)
- **authorDate_s**: date for the author (MARC 100d)
- **authorFuller_txt_en**: fuller form of the name (MARC 100q)
- **authorsOther_txts_en**: list of other authors (MARC 700a)
- **title_txt_en**: title of the book (MARC 245ab)
- **responsibility_txt_en**: statement of responsibility (MARC 245c)
- **publisher_txt_en**: publisher name (MARC 260a)
- **urls_ss**: URLs (MARC 856u)
- **subjects_txts_en**: an array of subjects (MARC 650a)
- **subjectsForm_txts_en**: (MARC 650v)
- **subjectsGeneral_txts_en**: (MARC 650x)
- **subjectsChrono_txts_en**: (MARC 650y)
- **subjectsGeo_txts_en**: (MARC 650z)

The suffix added to each field (e.g. `_s` or `_txt_en`) are hints for Solr to pick the appropriate field type for each field (e.g. string vs text in English) as ingests the data. We will look closely into this in a later section.

Fetching data

To fetch data from Solr we make an HTTP request to the `select` handler. For example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*'
```

There are many parameters that we can pass to this handler to define what documents we want to fetch and what fields we want to fetch.

Selecting what fields to fetch

We can use the `fl` parameter to indicate what fields we want to fetch. For example to request the `id` and the `title_txt_en` of the documents we would use `fl=id,title_txt_en` as in the following example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*&fl=id,title_txt_en'
```

Note: When issuing the commands via cURL (as in the previous example) make sure that the fields are separated by a comma *without any spaces in between them*. In other words make sure the URL says `fl=id,title_txt_en` and not `fl=id, title_txt_en`. If the parameter includes spaces Solr will not return any results and it will give you a cryptic error message.

Try adding and removing some other fields to this list, for example, `fl=id,author_txt_en,title_txt_en` or `fl=id,title_txt_en,author_txt_en,subjects_txts_en`

Filtering the documents to fetch

In the previous examples you might have seen an inconspicuous `q=*` parameter in the URL. The `q` (query) parameter tells Solr what documents to retrieve. This is somewhat similar to the `WHERE` clause in a SQL SELECT query.

If we want to retrieve all the documents we can just pass `q=*`. But if we want to filter we can use the following syntax: `q=field:value` to filter documents where a specific field has a particular value. For example, to include only documents where the `title_txt_en` has the word "teachers" we would use `q=title_txt_en:teachers`:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en,author_txt_en&q=title_txt_en:teachers'
```

We can request filter by many different fields, for example to request documents where the `title_txt_en`

includes the word "teachers" **or** the `author_txt_en` includes the word "Alice" we would use `q=title_txt_en:teachers author_txt_en:Alice`

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,author_txt_en&q=title_txt_en:teachers+author_txt_en:Alice'
```

As we saw in the previous example, by default, Solr searches for either of the terms. If we want to force that both conditions are matched we must explicitly use the `AND` operator in the `q` value as in `q=title_txt_en:teachers AND author_txt_en:Alice` Notice that the `AND` operator **must be in uppercase**.

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,author_txt_en&q=title_txt_en:teachers+AND+author_txt_en:Alice'
```

Now let's try something else. Let's issue a search for books where the title says "school teachers" using `q=title_txt_en:"school teachers"`

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:"school+teachers"'

  # the results will include
  # {
  #   "id":"00010001",
  #   "title":["... : strategies for middle and high school teachers /"]},
  # {
  #   "id":"00020575",
  #   "title":["... Sunday school teachers ... /"]},
  # {
  #   "id":"00011238",
  #   "title":["... solutions for middle and high school teachers /"]}]
  #
```

Notice how all three results have the term "school teachers" somewhere on the title. Now let's issue a slightly different query using `q=title_txt_en:"school teachers"~3` to indicate that we want the words "school" and "teachers" to be present in the `title_txt_en` but they can be a few words apart (notice the `~3`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:"school+teachers"~3'
```

The result for this query will include a new book with title "Aids to teachers of School chemistry" which includes both terms (school and teachers) but the order does not matter as long as they are close to each other.

One other thing. When searching multi-word keywords for a given field make sure the keywords are surrounded by quotes, for example make sure to use `q=title_txt_en:"school teachers"` and not `q=title_txt_en:school teachers`. The later will execute a search for "school" in the `title_txt_en` field and "teachers" in the `_text_` field.

You can validate this by running the query and passing the `debugQuery` flag and seeing the `parsedquery` value. For example in the following command we surround both search terms in quotes:

```
$ curl -s 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en&debugQuery=on&q=title_txt_en:"school+teachers"' | grep parsedquery

  # will show
  # "parsedquery":"PhraseQuery(title_txt_en:\"school teachers\")",
  #
```

notice that the `parsedQuery` shows that Solr is searching for, as we would expect, both words in the `title_txt_en` field.

Now let's look at the `parsedQuery` when we don't surround the search terms in quotes:

```
$ curl -s 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en&debugQuery=on&q=title_txt_en:school+teachers' | grep parsedquery

  # will show
  # "parsedquery":"title_txt_en:school _text_:teachers",
  #
```

notice that Solr searched for the word "school" in the `title_txt_en` field but searched for the word "teachers" on the `_text_` field. Certainly not what we were expecting. We'll elaborate in a later section on the significance of the `_text_` field but for now make sure to surround in quotes the search terms when issuing multi word searches.

One last thing to notice is that Solr returns results paginated, by default it returns the first 10 documents that match the query. We'll see later on this tutorial how we can request a large page size (via the `rows` parameter) or another page (via the `start` parameter). But for now just notice that at the top of the results Solr always tells us the total number of results found:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=title:education&fl=id,title'

  #
  # response will include
  #   "response":{"numFound":101,"start":0,"docs":[
  #
```

Getting facets

When we issue a search, Solr is able to return facet information about the data in our core. This is a built-in feature of Solr and easy to use, we just need to include the `facet=on` and the `facet.field` parameter with the name of the field that we want to facet the information on.

For example, to search for all documents with title "education" (`q=title_txt_en:education`) and retrieve facets (`facet=on`) based on the subjects (`facet.field=subjects_txts_en_str`) we'll use a query like this:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,author_txt_en&q=title_txt_en:education&facet=on&facet.field=subjects_txts_en_str'

  # response will include something like this
  #
  # "facet_counts":{
  #   "facet_queries":{},
  #   "facet_fields":{
  #     "subjects_str":[
  #       "Education",12,
  #       "Education, Higher",10,
  #       "Universities and colleges",6,
  #       "Educational equalization",4,
  #       "Women",4,
  #
```

You might have noticed that we used the "string" version of the subjects (`subjects_txts_en_str`) rather than the "text" version of them (`subjects_txts_en`). This is because using the "text" version would have generated facets based on the *tokenized* version of the subjects rather than the original subjects. We will review tokenization in a later section but if you want to see the difference run this command and notice the facets returned:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,author_txt_en&q=title_txt_en:education&facet=on&facet.field=subjects_txts_en'
```

Updating documents

To update a document in Solr you have two options. One option is to post the data for that document again to

Solr and let Solr overwrite the old document with the new data. The key for this to work is to provide *the same ID in the new data* as the ID of an existing document.

For example, if we query the document with ID `00000034` we would get:

```
$ curl "http://localhost:8983/solr/bibdata/select?q=id:00000034"

  # "response":{"numFound":1,"start":0,"docs":[
  # {
  #   "id":"00000034",
  #   "author_txt_en":"Burrows Brothers Company, Cleveland.",
  #   "title_txt_en":"A catalogue of the best books in every....",
  #   "publisher_txt_en":"Cleveland,",
  #   "subjects_txts_en":["Booksellers' catalogs"],
  #   "subjectsGeo_txts_en":["United States"],
  #   "subjects_txts_en_str":["Booksellers' catalogs"],
  #   "_version_":1624648963022913536,
  #   "subjectsGeo_txts_en_str":["United States"]}]
  # }}
  #
```

If we post to Solr a new document with the **same ID** Solr will **overwrite** the existing document with the new data. Below is an example of how to update this document with new JSON data using `curl` to post the data to Solr:

```
$ curl -X POST --data '[{"id":"00000034","title_txt_en":"the new title"}]'
"http://localhost:8983/solr/bibdata/update?commit=true"
```

Out of the box Solr supports multiple input formats (JSON, XML, CSV), section Uploading Data with Index Handlers in the Solr guide provides more details out this.

If we query for the document with ID `00000034` again we will see the new data and notice that the fields that we did not provide in the update are now gone from the document, that's because Solr overwrote the old document with ID `00000034` with our new data that included only two fields (`id` and `title_txt_en`).

```
$ curl "http://localhost:8983/solr/bibdata/select?q=id:00000034"

  # "response":{"numFound":1,"start":0,"docs":[
  # {
  #   "id":"00000034",
  #   "title_txt_en":"the new title"
  # }]}
  #
```

The second option to update a document is to update only parts of a document, but that is out of scope for this tutorial. The Solr Guide provides information on how this works.

Deleting documents

To delete all documents for the `bibdata` core we can submit a request to Solr's `update` endpoint (rather than the `select` endpoint) with a command like this:

```
$ curl "http://localhost:8983/solr/bibdata/update?commit=true" --data '<delete><query>id:00020424</query>
</delete>'
```

The body of the request (`--data`) indicates to Solr that we want to delete a specific document (notice the `id:00020424` query).

We can also pass a less specific query like `title_txt_en:teachers` to delete all documents where the title includes the word "teachers" (or a variation of it). Or we can delete *all documents* with a query like `*:*`.

Be aware that even if you delete all documents from a Solr core the schema and the core's configuration will remain intact. For example, the fields that were defined are still available even if no documents exist in the core anymore.

If you want to delete the entire core (documents, schema, and other configuration associated with it) you can use the Solr delete command instead:

```
$ cd ~/solr-7.4.0/bin
$ ./solr delete -c bibdata
```

be aware that you will need to re-create the core if you want to re-import data to it.

PART II: SCHEMA

The schema in Solr is the definition of the *field types* and *fields* configured for a given core.

Field Types are the building blocks to define fields in our schema. Examples of field types are: `binary`, `boolean`, `pfloat`, `string`, `text_general`, and `text_en`. These are akin to the "field types" that are supported in a relational database like MySQL but, as we will see later, they are far more configurable than what you can do in a relational database.

There are three kind of fields that can be defined in a Solr schema:

- **Fields** are the specific fields that you define for your particular core. Fields are based of a "field type", for example, we might define field `title` based on the `string` field type and a field `price` base of the `pfloat` field type.
- **dynamicFields** are field patterns that we define to automatically create new fields when the data submitted to Solr matches the given pattern. For example, we can define that if receive data for a field that ends with `_txt` the field will be create it as a `text_general` field type.
- **copyFields** are instructions to tell Solr how to automatically copy the value given for one field to another field. This is useful if we want to do and store different transformation on the values given to us. For example, we might want to remove punctuation characters for searching but preserve them for display purposes.

Our newly created `bibdata` core already has a schema and you can view the definition through the Solr Admin web page via the Schema Browser Screen or by exploring the `managed-schema` file via the Files Screen.

You can also view this information with the Schema API as shown in the following example. The response will be rather long and it will be organized in four categories: `fieldTypes`, `fields`, `dynamicFields`, and `copyFields` as shown below:

```
$ curl localhost:8983/solr/bibdata/schema

  # {
  #   "responseHeader": {"status": 0, "QTime": 2},
  #   "schema": {
  #     "fieldTypes":   [lots of field types defined],
  #
  #     "fields":       [lots of fields defined],
  #
  #     "dynamicFields":[lots of dynamic fields defined],
  #
  #     "copyFields":   [a few copy fields defined]
  #   }
  # }
  #
```

The advantage of the Schema API is that it allows you to view *and update* the information programatically which is useful if you need to recreate identical Solr cores without manually configuring each field definition.

You can request information about each of these categories individually in the Schema API with the following commands (notice that combined words like `fieldTypes` and `dynamicFields` are *not* capitalized in the URLs below):

```
$ curl localhost:8983/solr/bibdata/schema/fieldtypes
$ curl localhost:8983/solr/bibdata/schema/fields
$ curl localhost:8983/solr/bibdata/schema/dynamicfields
$ curl localhost:8983/solr/bibdata/schema/copyfields
```

Notice that unlike a relational database, where only a handful field types are available to choose from (e.g. integer, date, boolean, char, and varchar) in Solr there are lots of predefined field types available out of the

box, and each of them with its own configuration.

Note for Solr 4.x users: In Solr 4 the default mechanism to update the schema was by editing the file `schema.xml`. Starting in Solr 5 the default mechanism is through the "Managed Schema Definition" which uses the Schema API to add, edit, and remove fields. There is a `managed-schema` file with the same information as `schema.xml` but you are not supposed to edit this file. See section "Managed Schema Definition in SolrConfig" in the Solr Reference Guide 5.0 (PDF) for more information about this.

Fields in our schema

You might be wondering how did the fields like `id`, `title_txt_en`, `author_txt_en`, `subjects_txts_en`, and `subjects_txts_en_str` in our `bibdata` core were created if we never explicitly defined them.

Solr automatically created most of these fields when we imported the data from the `books.json` file. If you look at a few of the elements in the `books.json` file you'll recognize that they match most of the fields defined in our schema. Below is the data for one of the records in our sample data:

```
{
  "id":"00000018",
  "author_txt_en":"Tarbell, H. S.",
  "authorDate_s":"1838-1904.",
  "authorFuller_txt_en":"(Horace Sumner),",
  "authorsOther_txts_en":["Tarbell, Martha,"],
  "title_txt_en":"The complete geography.",
  "publisher_txt_en":"New York,",
  "subjects_txts_en":["Geography"]
}
```

The process that Solr follows when a new document is ingested into Solr is more or less as follows:

- If there is an exact match for a field being ingested and the fields defined in the Schema then Solr will use the definition in the Schema to ingest the data. This is what happened for the `id` field. Our JSON data has an `id` field and so does the Schema, therefore Solr stored the `id` value in the `id` field as indicated in the Schema (i.e. as single-value string.)
- If there is no exact match in the Schema then Solr will look at the **dynamicFields** definitions to see if the field can be indexed with some predefined settings. This is what happened with the `title_txt_en` field. Because there is not `title_txt_en` definition in the Schema Solr used the dynamic field definition for `*_txt_en` that indicated that the value should be indexed using the text in English (`text_en`) field definition.
- If no match is found in the dynamic fields either Solr will make a guess on what is the best type to use based on the data for this field in the source document. This is what happened with the `subjects_txts_en` field (notice that it ends with `_txts_en` rather than `_txt_en`). In this instance Solr guessed and created the field as `text_general` multi-valued. Additionally, Solr has built-in logic to automatically create a string representation of these text fields and hence you'll also see a `subjects_txts_en_str` in the schema.

In the following sections we are going to drill down into some of specifics of the fields and dynamic field definitions that are configured in our Solr core.

Note: You can disable automatic field creation if you don't want this behavior. Keep in mind that if automatic field creation is disabled Solr will *refuse* to import any documents with fields not defined in the schema. For more information about the different way in which the Solr Schema can be configured check out this blog post.

Field: id

Let's look at the details the `id` field in our schema

```
$ curl localhost:8983/solr/bibdata/schema/fields/id
```

```
  #
  # Will return something like this
  # {
  #   "responseHeader":{...},
  #   "field":{
  #    "name":"id",
  #    "type":"string",
  #    "multiValued":false,
  #    "indexed":true,
  #    "required":true,
  #    "stored":true
  #   }
  # }
  #
```

Notice how the field is of type `string` but also it is marked as not multi-value, to be indexed, required, and stored.

The type `string` has also its own definition which we can view via:

```
$ curl localhost:8983/solr/bibdata/schema/fieldtypes/string

  # {
  #   "responseHeader":{...},
  #   "fieldType":{
  #     "name":"string",
  #     "class":"solr.StrField",
  #     "sortMissingLast":true,
  #     "docValues":true
  #   }
  # }
  #
```

In this case the `class` points to an internal Solr class (`solr.StrField`) that will be used to handle values of the string type.

Field: title_txt_en

Now let's look at a more complex field and field type. If we look for a definition for the `title_txt_en` Solr will report that we don't have one:

```
$ curl localhost:8983/solr/bibdata/schema/fields/title_txt_en

  # {
  #   "responseHeader":{...
  #   "error":{
  #     "metadata":[
  #      "error-class","org.apache.solr.common.SolrException",
  #      "root-error-class","org.apache.solr.common.SolrException"],
  #    "msg":"No such path /schema/fields/title_txt_en",
  #    "code":404}}
  #
```

However, if we look at the dynamic field definitions we'll notice that there is one for fields that end in `_txt_en`:

```
$ curl localhost:8983/solr/bibdata/schema/dynamicfields/*_txt_en

  # {
  #   "responseHeader":{...
  #   "dynamicField":{
  #     "name":"*_txt_en",
  #     "type":"text_en",
  #     "indexed":true,
  #     "stored":true}}
  #
```

This tells Solr that any field name in the source data that does not already exist in Solr and that ends in `_txt_en`

should be created as a field of type `text_en`. That looks innocent enough, so let's look closer to see what field type `text_en` means:

```
$ curl localhost:8983/solr/bibdata/schema/fieldtypes/text_en

  # {
  #   "responseHeader":{...}
  #   "fieldType":{
  #     "name":"text_en",
  #     "class":"solr.TextField",
  #     "positionIncrementGap":"100",
  #     "indexAnalyzer":{
  #       "tokenizer":{
  #         "class":"solr.StandardTokenizerFactory"
  #       },
  #       "filters":[
  #         { "class":"solr.StopFilterFactory" ... },
  #         { "class":"solr.LowerCaseFilterFactory" },
  #         { "class":"solr.EnglishPossessiveFilterFactory" },
  #         { "class":"solr.KeywordMarkerFilterFactory" ... },
  #         { "class":"solr.PorterStemFilterFactory" }
  #       ]
  #     },
  #     "queryAnalyzer":{
  #       "tokenizer":{
  #         "class":"solr.StandardTokenizerFactory"
  #       },
  #       "filters":[
  #         { "class":"solr.SynonymGraphFilterFactory" ... },
  #         { "class":"solr.StopFilterFactory" ... },
  #         { "class":"solr.LowerCaseFilterFactory" },
  #         { "class":"solr.EnglishPossessiveFilterFactory" },
  #         { "class":"solr.KeywordMarkerFilterFactory" ... },
  #         { "class":"solr.PorterStemFilterFactory" }
  #       ]
  #     }
  #   }
  # }
```

This is obviously a much more complex definition than the ones we saw before. Although the basics are the same (e.g. the field type points to class `solr.TextField`) notice that there are two new sections `indexAnalyzer` and `queryAnalyzer` for this field type. We will explore those in the next section.

Note: The fact that the Solr schema API does not show dynamically created fields (like `title_txt_en`) is baffling, particularly since they do show in the Schema Browser Screen of the Solr Admin screen. This has been a known issue for many years as shown in this Stack Overflow question from 2010 in which one of the answers suggests using the following command to list all fields, including those created via `dynamicField` definitions: `curl localhost:8983/solr/bibdata/admin/luke?numTerms=0`

Analyzers, Tokenizers, and Filters

The `indexAnalyzer` section defines the transformations to perform *as the data is indexed* in Solr and `queryAnalyzer` defines transformations to perform *as we query for data* out of Solr. It's important to notice that the output of the `indexAnalyzer` affects the terms *indexed*, but not the value *stored*. The Solr Reference Guide says:

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

When a value is *indexed* for a particular field the value is first passed to a `tokenizer` and then to the `filters` defined in the `indexAnalyzer` section for that field type. Similarly, when we *query* for a value in a given field the value is first processed by a `tokenizer` and then by the `filters` defined in the `queryAnalyzer` section for that field.

If we look again at the definition for the `text_en` field type we'll notice that "stop words" (i.e. words to be ignored) are handled at index and query time (notice the `StopFilterFactory` filter appears in the `indexAnalyzer` and the `queryAnalyzer` sections.) However, notice that "synonyms" will only be applied at query time since the filter `SynonymGraphFilterFactory` only appears on the `queryAnalyzer` section.

We can customize field type definitions to use different filters and tokenizers via the Schema API which we will discuss later on this tutorial.

Tokenizers

For most purposes we can think of a tokenizer as something that splits a given text into individual tokens or words. The Solr Reference Guide defines Tokenizers as follows:

```
Tokenizers are responsible for breaking
field data into lexical units, or tokens.
```

For example if we give the text "hello world" to a tokenizer it might split the text into two tokens like "hello" and "word".

Solr comes with several built-in tokenizers that handle a variety of data. For example if we expect a field to have information about a person's name the Standard Tokenizer might be appropriated for it. However, for a field that contains e-mail addresses the UAX29 URL Email Tokenizer might be a better option.

I believe you can only have one tokenizer per analyzer

Filters

Whereas a `tokenizer` takes a string of text and produces a set of tokens, a `filter` takes a set of tokens, process them, and produces a different set of tokens. The Solr Reference Guide says that

```
in most cases a filter looks at each token in the stream sequentially
and decides whether to pass it along, replace it or discard it.
```

Notice that unlike tokenizers, whose job is to split text into tokens, the job of filters is a bit more complex since they might replace the token with a new one or discard it altogether.

Solr comes with many built-in Filters that we can use to perform useful transformations. For example the ASCII Folding Filter converts non-ASCII characters to their ASCII equivalent (e.g. "México" is converted to "Mexico"). Likewise the English Possessive Filter removes singular possessives (trailing 's) from words. Another useful filter is the Porter Stem Filter that calculates word stems using English language rules (e.g. both "jumping" and "jumped" will be reduced to "jump".)

Putting it all together

When we looked at the definition for the `text_en` field type we noticed that at *index time* several filters were applied (`StopFilterFactory`, `LowerCaseFilterFactory`, `EnglishPossessiveFilterFactory`, `KeywordMarkerFilterFactory`, and `PorterStemFilterFactory`.)

That means that if we *index* the text "The Television is Broken!" in a `text_en` field the filters defined in the `indexAnalyzer` will transform this text into two tokens: "televis", and "broken". Notice how the tokens were lowercased, the stop words ("the" and "is") dropped, and only the stem of the word "television" was indexed.

Likewise, the definition for `text_en` included the additional filter `SynonymGraphFilter` at *query time*. So if we were to *query* for the text "The TV is Broken!" Solr will run this text through the filters indicated in the `queryAnalyzer` section and generate the following tokens: "televis", "tv", and "broken". Notice that an additional transformation was done to this text, namely, the word "TV" was expanded to its synonyms. This is because the

`queryAnalyzer` uses the `SynonymGraphFilter` and a standard Solr configuration comes with those synonyms predefined in the `synonyms.txt` file.

The Solr Analysis Screen in the Solr Admin tool is a great way to see a particular text is either indexed or queried by Solr *depending on the field type*. Here is a few examples to try:

- Enter "The quick brown fox jumps over the lazy dog" in the "Field Value (*index*)", select `string` as the field type and see how is indexed. Then select `text_general` and click "Analyze Values" to see how it's indexed. Lastly, select `text_en` and see how it's indexed. You might want to uncheck the "Verbose output" to see the differences more clearly.
- With the text still on the "Field Value (*index*)" text box, enter "The quick brown fox jumps over the LAZY dog" on the "Field Value (*query*)" and try the different field types (`string/text_general/text_en`) again to see how each of them shows different matches.
- Try changing the text on the "Field Value (*query*)" text box to "The quick brown foxes jumped over the LAZY dogs". Compare the results using `text_general` versus `text_en`.
- Now enter "The TV is broken!" on the "Field Value (*index*)" text box, clear the "Field Value (*query*)" text box, select `text_en`, and see how the value is indexed. Then do the reverse, clear the indexed value and enter "The TV is broken!" on the "Field Value (*query*)" text box and notice synonyms being applied.
- Now enter "The TV is broken!" on the "Field Value (*index*)" text box and "the television is broken" on the "Field Value (*query*)". Notice how they are matched because the use of synonyms applied for `text_en` fields.
- Now enter "The TV is broken!" on the "Field Value (*index*)" text box and clear the "Field Value (*query*)" text box, select `text_general` and notice how the stop words were not removed because we are not using English specific rules.

Handling text in Chinese, Japanese, and Korean (optional)

If your data has text in Chinese, Japanese, or Korean (CJK) Solr has built-in support for searching text in these languages using the proper transformations. Just as Solr uses different transformation when using field type `text_en` instead of `text_general` Solr applies different rules when using field type `text_cjk`.

You can see the definition of this field type with the following command. Notice how there are two new filters (`CJKWidthFilterFactory` and `CJKBigramFilterFactory`) that are different from what we saw in the `text_en` definition.

```
$ curl localhost:8983/solr/bibdata/schema/fieldtypes/text_cjk

  # ...
  # "fieldType":{
  #   "name":"text_cjk",
  #   "class":"solr.TextField",
  #   "positionIncrementGap":"100",
  #   "analyzer":{
  #     "tokenizer":{
  #       "class":"solr.StandardTokenizerFactory"},
  #     "filters":[
  #       {"class":"solr.CJKWidthFilterFactory"},
  #       {"class":"solr.LowerCaseFilterFactory"},
  #       {"class":"solr.CJKBigramFilterFactory"}]}}}
  #
```

If you go to the Analysis Screen again and enter "胡志明" (Ho Chi Minh) as the "Field Value (index)", select `text_general` as the FieldType and analyse the values you'll notice how Solr calculated three tokens ("胡", "志", and "明") which is incorrect in Chinese. However, if you select `text_cjk` and analyze the values again you'll notice that you'll end with two tokens ("胡志" and "志明") thanks to the `CJKBigramFilterFactory` and that is the expected behavior for text in chinese.

The data for this section was taken from this blog post. Although the technology referenced in the blog post is a bit dated, the basic concepts explained are still relevant, particularly if you, like me, are not a CJK speaker.

Stored vs indexed fields (optional)

There are two properties on a Solr field that control whether its values are `stored`, `indexed`, or both. Fields that are *stored but not indexed* can be fetched once a document has been found, but you cannot search by those fields (i.e. you cannot reference them in the `q` parameter). Fields that are *indexed but not stored* are the reverse, you can search by them but you cannot fetch their values once a document has been found (i.e. you cannot reference them in the `fl` parameter). Technically is also possible to add a field that is neither stored nor indexed but that's beyond the scope of this tutorial.

For example, let's say that we add the following fields to our Solr core:

- f_stored: a field stored but not indexed
- f_indexed: a field indexed but not stored
- f_both: a field both indexed and stored

Create `f_stored` field:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
    "name":"f_stored",
    "type":"text_general",
    "multiValued":false,
    "stored":true,
    "indexed":false
  }
}' http://localhost:8983/solr/bibdata/schema
```

Create `f_indexed` field:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
    "name":"f_indexed",
    "type":"text_general",
    "multiValued":false,
    "stored":false,
    "indexed":true
  }
}' http://localhost:8983/solr/bibdata/schema
```

Create `f_both` field:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
    "name":"f_both",
    "type":"text_general",
    "multiValued":false,
    "stored":true,
    "indexed":true
  }
}' http://localhost:8983/solr/bibdata/schema
```

And now let's add a document with data for these new fields:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '[{
  "id": "f_demo",
  "f_stored": "stored",
  "f_indexed": "indexed",
  "f_both": "both",
  "name": "test stored vs indexed fields"
}]' http://localhost:8983/solr/bibdata/update?commit=true
```

If we query for this document notice how `f_stored` and `f_both` will be fetched but *not* `f_indexed` (even though we list it in the `fl` parameter) because `f_indexed` is not stored.

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=id:f_demo&fl=id,f_indexed,f_stored,f_both'

  # "response":{"numFound":1,"start":0,"docs":[
  # {
  #   "id":"f_demo",
  #   "f_stored":"stored",
  #   "f_both":"both"}]
  # }}
```

Keep in mind that we can search by `f_indexed` because it is indexed:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=f_indexed:indexed'

  # will find the document, but again, the value of field f_indexed
  # will not be included in results.
```

Notice that even though we are able to fetch the value for the `f_stored` field we cannot use it for searches:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=f_stored:stored'

  # will return no results
  #   "response":{"numFound":0,"start":0,"docs":[]
```

Lastly, our indexed and stored field (`f_both`) can be searched for and fetched.

There are many reasons to toggle the stored and indexed properties of a field. For example, perhaps we want to store a complex object as string in Solr so that we can display it to the user but we really don't want to index its values. Conversely, perhaps we want to create a field with a combination of values and search on that field but we don't want to display it to the users (the default `_text_` field in our schema is such an example).

Customizing our schema

So far we have only worked with the fields that were automatically added to our `bibdata` core as we imported the data. Because the fields in our source data had suffixes (`_s`, `_txt_en`) that match with the default `dynamicField` definitions in a standard Solr installation most of our fields were created with the proper field type except, as we saw earlier, the `_txts_en` field which was created as a `text_general` field rather than at `text_en` field.

Also, although it's nice that we can do sophisticated searches by title (because it is a `text_en` field) we could not sort the results by this field because it's a tokenized field, technically we can sort by it but the results will not be what we expect.

Let's customize our schema a little bit to get the most out of Solr.

Recreating our Solr core

Let's begin by recreating our Solr core so that we have a clean slate.

```
$ cd ~/solr-7.4.0/bin
$ ./solr delete -c bibdata
$ ./solr create -c bibdata
```

Before we import again the data in our `books.json` file we are going to add a few field definitions to the Schema to make sure the data is ingested in the way that we want to.

Handling `_txts_en` fields

The first thing we'll do is add a new `dynamicField` definition to account for multi-value text fields in English for fields that end with `_txts_en` in our JSON data:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-dynamic-field":{
     "name":"*_txts_en",
     "type":"text_en",
     "multiValued":true}
}' http://localhost:8983/solr/bibdata/schema
```

this will make sure Solr indexes these fields `text_en` rather than the default `text_general` that it used when we did not have an `dynamicField` to account for them.

Customizing the title field

Secondly we'll ask Solr to store a string version of the title (in addition to the text version) so we can sort results by title. To do this we'll add a `copy-field` directive to our Schema to copy the value of the `title_txt_en` to another field (`title_s`). This way we'll have a text version for searching and a string version for sorting.

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-copy-field":[
    {
      "source":"title_txt_en",
      "dest":[ "title_s" ]
    }
  ]
}' http://localhost:8983/solr/bibdata/schema
```

Customizing the author fields

Right now we have two separate fields for author information (`author_txt_en` for the main author and `authorsOther_txts_en` for additional authors) which means that if we want to find books by a particular author we have to issue a query against two separate fields: `author_txt_en:"Sarah" OR authorsOther_txts_en:"Sarah"`

Let's use a `copy-field` directive to have Solr automatically combine the main author and additional authors into a new field. Notice that the new field `authorsAll_txts_en` matches the `dynamicField` directive that we just created, meaning that it will be indexed as `text_en` multi-valued.

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-copy-field":[
    {
      "source":"author_txt_en",
      "dest":[ "authorsAll_txts_en" ]
    },
    {
      "source":"authorsOther_txts_en",
      "dest":[ "authorsAll_txts_en" ]
    }
  ]
}' http://localhost:8983/solr/bibdata/schema
```

We could also add another `copy-field` directive to copy the main author from a text field (`author_txt_en`) to a string field (`author_s`) so that we can sort search results by author. But we'll skip that for now.

Customizing the subject and publisher fields

Another customization that we'll do is create a string representation of the `subjects_txts_en` field so that we can use subjects as facets (remember that facets require string fields).

This string version of the field is something that we got for free when Solr automatically guessed the field type to

use for this field (remember that Solr created an additional `subjects_txts_en_str` for us). Now that we are handling the field explicitly with our `*_txts_en` `dynamicField` definition we need to tell Solr explicitly to create this extra field:

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-copy-field":[
    {
      "source":"subjects_txts_en",
      "dest": "subjects_ss",
      "maxChars": "100"
    }
  ]
}' http://localhost:8983/solr/bibdata/schema
```

While we are at it, let's also create a string representation of the `publisher_txt_en` so that we can use publisher in the facets too.

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-copy-field":[
    {
      "source":"publisher_txt_en",
      "dest": "publisher_s",
      "maxChars": "100"
    }
  ]
}' http://localhost:8983/solr/bibdata/schema
```

Testing our changes

Now that we have configured our schema with a few specific field definitions let's re-import the data so that fields are indexed using the new configuration.

```
$ cd ~/solr-7.4.0/bin
$ ./post -c bibdata books.json
```

Testing changes to the title field

Now that we have a string version of the title field is possible for us to sort our search results by this field, for example, let's search for books that have the word "water" in the title (`q=title_txt_en:water`) and sort them by title (`sort=title_s+asc`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:water&sort=title_s+asc'

  #
  # response will include
  #   ...
  #   {"title_txt_en":"A unit of water, a unit of time : Joel White's last boat"},
  #   {"title_txt_en":"Annotated list of birds of ... inland water-fowl ..."},
  #   {"title_txt_en":"Boundary waters canoe camping /"},
  #   {"title_txt_en":"Clean coastal waters : understand.."},
  #   ...
  #
```

notice that the result are sorted alphabetically by title because we are using the string version of the field (`title_s`) for sorting. Try and see what the results look like if you sort by the text version of the title (`title_txt_en`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en&q=title_txt_en:water&sort=title_txt_en+asc'
```

The results in this case will not look correct because Solr will be using the tokenized value of the `title_txt_en` field to sort rather than the string version.

Testing changes to the author field

Take a look at the data for this particular book that has many authors and notice how the `authorsAll_txts_en` field has the combination of `author_txt_en` and `authorsOther_txts_en` even though our source data didn't have an `authorsAll_txts_en` field:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=id:00009214&fl=id,author*'

  #
  # {
  #   "id":"00009214",
  #    "author_txt_en":"Everett, Barbara,",
  #    "authorsOther_txts_en":["Gallop, Ruth,"]
  #    "authorsAll_txts_en":["Everett, Barbara,", "Gallop, Ruth,"],
  # }
  #
```

Likewise, let's search for books authored by "Gallop" using our new `authorsAll_txts_en` field (`q=authorsAll_txts_en:Gallop`) and notice how this document will be on the results regardless of whether Ruth Gallop is the main author or an additional author.

What are others doing

There are lots of pre-defined dynamic fields in a standard Solr installation as you can see in the `managed-schema` file under the Files Screen. You can also see at the `schema.xml` for some of the open source projects that are using Solr.

Here are a few examples:

- Brown University Library Catalog (a Blacklight app): https://github.com/Brown-University-Library/bul-search/blob/master/solr_conf/blacklight-core/conf/schema.xml
- Penn State ScholarSphere (a Hydra/SamVera app): https://github.com/psu-stewardship/scholarsphere/blob/develop/solr/config/schema.xml
- Princeton University Library (a Blacklight app): https://github.com/pulibrary/pul_solr/blob/master/solr_configs/catalog-production/conf/schema.xml

PART III: SEARCHING

When we issue a search to Solr we pass the search parameters in the query string. In previous examples we passed values in the `q` parameter to indicate the values that we want to search for and `fl` to indicate what fields we want to retrieve. For example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*&fl=id,title_txt_en'
```

In some instances we passed rather sophisticated values for these parameters, for example we used `q=title_txt_en:"school teachers"~3` when we wanted to search for books with the words "school" and "teachers" in the title within a few word words of each other.

The components in Solr that parse these parameters are called Query Parsers. Their job is to extract the parameters and create a query that Lucene can understand. Remember that Lucene is the search engine underneath Solr.

Query Parsers

Out of the box Solr comes with three query parsers: Standard, DisMax, and Extended DisMax (eDisMax). Each of them has its own advantages and disadvantages.

- The Standard query parser (aka the Lucene Parser) is the default parser and is very powerful, but it's rather unforgiving if there is an error in the query submitted to Solr. This makes the Standard query parser a poor choice if we want to allow user entered queries, particular if we allow queries with expressions like `AND` or `OR` operations.
- The DisMax query parser (DisMax) on the other hand was designed to handle user entered queries and is very forgiving on errors when parsing a query, however this parser only supports simple query expressions.
- The Extended DisMax (eDisMax) query parser is an improved version of the DisMax parser that is also very forgiving to errors when parsing user entered queries and like the Standard query parser supports complex query expressions.

One key difference among these parsers is that they recognize different parameters. For example, the *DisMax* and *eDisMax* parsers supports a `qf` parameter to specify what fields should be searched for but this parameter is not supported by the *Standard* parser.

The rest of the examples in this section are going to use the eDisMax parser, notice the `defType=edismax` in our queries to Solr to make this selection. As we will see later on this tutorial you can also set the default query parser of your Solr core to use eDisMax by updating the `defType` parameter in your `solrconfig.xml` so that you don't have to explicitly set it on every query.

Basic searching in Solr

The number of search parameters that you can pass to Solr is rather large and, as we've mentioned, they also depend on what query parser you are using.

To see a list a comprehensive list of the parameters that apply to all parsers take a look at the Common Query Parameters and the Standard Query Parser sections in the Solr Reference Guide.

Below are some of the parameters that are supported by all parsers:

- `defType`: Query parser to use (default is `lucene`, other possible values are `dismax` and `edismax`)
- `q`: Search query, the basic syntax is `field:"value"`.
- `sort`: Sorting of the results (default is `score desc`, i.e. highest ranked document first)
- `rows`: Number of documents to return (default is `10`)

- `start`: Index of the first document to result (default is `0`)
- `fl`: List of fields to return in the result.
- `fq`: Filters results without calculating a score.

Below are a few sample queries to show these parameters in action. Notice that spaces are URL encoded as `+` in the commands below, you do not need to encode them if you are submitting these queries via the Solr Admin interface in your browser.

- Retrieve the first 10 documents where the `title_txt_en` includes the word "washington" (`q=title_txt_en:washington`)

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=title_txt_en:washington'
```

- The next 15 documents for the same query (notice the `start=10` and `rows=15` parameters)

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=title_txt_en:washington&start=10&rows=15'
```

- Retrieve the `id` and `title_txt_en` (`fl=id,title_txt_en`) where the title includes the words "women writers" but allowing for a word in between e.g. "women nature writers" (`q=title_txt_en:"women writers"~1`) Technically the `~N` means "N edit distance away" (See Solr in Action, p. 63).

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=title_txt_en:"women+writers"~1&fl=id,title_txt_en'
```

- Documents that have additional authors (`q=authorsOther_txt_en:*`), the `*` means "any value".

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title,author_txt_en,authorsOther_txts_en&q=authorsOther_txts_en:*'
```

- Documents that do *not* have additional authors (`q=NOT authorsOther_txt_en:*`)

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,author_txt_en&q=NOT+authorsOther_txt_en:*'
```

- Documents where at least one of the subjects has a word that starts with "com" (`q=subjects_txts_en:com*`)

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title,subjects_txts_en&q=subjects_txts_en:com*'
```

- Documents where title include "story" *and* at least one of the subjects is "women" (`q=title_txt_en:story AND subjects_txts_en:women` notice that both search conditions are indicated in the `q` parameter) Again, please notice that the `AND` operator **must be in uppercase**.

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,subjects_txts_en&q=title_txt_en:story+AND+subjects_txts_en:women'
```

- Similar to the previous query, documents where title include "story" (`q=title_txt_en:story`) *and* at least one of the subjects has the word "women" (`fq=subjects_txts_en:women`) but *without considering the subject in the ranking of the results* (notice that subjects are filtered via the `fq` parameter in this example)

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,subjects_txts_en&q=title_txt_en:story&fq=subjects_txts_en:women'
```

- Documents where title *includes* the word "american" but *does not include* the word "america" (`q=title_txt_en:history AND -title_txt_en:america`)

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:history+AND+-
title_txt_en:america'
```

The Solr Reference Guide and this tutorial are good places to check for quick reference on the query syntax.

q and fq parameters

Solr supports two different parameters to filter results in a query. One is the Query `q` parameter that we've been using in all our examples. The other is the Filter Query `fq` parameter that we introduced in the last query. Both parameters can be used to filter the documents to return in a query, but there is a key difference between them: `q` calculates scores for the results whereas `fq` does not.

In Solr in Action (p. 211) the authors say:

```
So what is the difference between the q and fq parameters?

fq serves a single purpose: to limit your results to a set
of matching documents.

The q parameter, in contrast, serves two purposes:
  * To limit your results to a set of matching documents
  * To supply the relevancy algorithm with a list of terms
    to be used for relevancy scoring
```

The reason this is important is because values filtered via `fq` can be cached and reused better by Solr in subsequent queries because they don't have a score assigned to them. The authors of Solr in Action recommend using the `q` parameter for values entered by the user and `fq` for values selected from a list (e.g. from a dropdown or a facet in an application)

Both `q` and `fq` use the same syntax for filtering documents (e.g. `field:value`). However you can only have one `q` parameter in a query but you can have many `fq` parameters. Multiple `fq` parameters are ANDed (you cannot specify an OR operation among them).

the qf parameter

The DisMax and eDisMax query parsers provide another parameter, Query Fields `qf`, that should not be confused with the `q` or `fq` parameters. The `qf` parameter is used to indicate the *list of fields* that the search should be executed on along with their boost values.

As we saw in a previous example if we execute a search on multiple fields and give each of them a different boost value the `qf` parameter makes this relatively easily as we can indicate the search terms in the `q` parameter (`q="george washington"`) and list the fields and their boost values separately (`qf=title_txt_en authorsAll_txts_en^10`). Remember to select the eDisMax parser (`defType=edismax`) when using the `qf` parameter.

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,authorsAll_txts_en&q="george+washington"&qf=title_txt_en+authorsAll_txts_en^20&defType=edismax
'
```

Boost values are arbitrary, you can use 1, 20, 789, 76.2, 1000, or whatever number you like, you can even use negative numbers (`qf=title_txt_en authorsAll_txts_en^-10`). They are just a way for us to hint Solr which fields we consider more important in a particular search.

debugQuery

Solr provides an extra parameter `debugQuery=on` that we can use to get debug information about a query. This is particularly useful if the results that you get in a query are not what you were expecting. For example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=title_txt_en:west+AND+authorsAll_txts_en:nancy&fl=id,title_txt_en,authorsAll_txts_en&defType=edismax&debugQuery=on
'

  # response will include
  # {
  #   "responseHeader":{...}
  #   "response":{...}
  #   "debug":{
  #     "rawquerystring":"title_txt_en:west AND authorsAll_txts_en:nancy",
  #     "querystring":"title_txt_en:west AND authorsAll_txts_en:nancy",
  #     "parsedquery":"+(+title_txt_en:west +authorsAll_txts_en:nanci)",
  #     "parsedquery_toString":"+(+title_txt_en:west +authorsAll_txts_en:nanci)",
  #     "explain":{
  #       ... tons of information here ...
  #     }
  #     "QParser":"ExtendedDismaxQParser",
  #   }
  # }
  #
```

Notice the `debug` property, inside this property there is information about:

- what value the server received for the search (`querystring`) which is useful to detect if you are not URL encoding properly the value sent to the sever
- how the server parsed the query (`parsedquery`) which is useful to detect if the syntax on the `q` parameter was parsed as we expected (e.g. remember the example earlier when we passed two words `school teachers` without surrounding them in quotes and the parsed query showed that it was querying two different fields `title_txt_en` for "school" and `_text_` for "teachers")
- you can also see that some of the search terms were stemmed (e.g. "nancy" was converted to "nanci")
- how each document was ranked (`explain`)
- what query parser (`QParser`) was used

Ranking of documents

When Solr finds documents that match the query it ranks them so that the most relevant documents show up first. You can provide Solr guidance on what fields are more important to you so that Solr consider this when ranking documents that matches a given query.

Let's say that we want documents where either the `title_txt_en` or the `author` have the word "west", we would use `q=title_txt_en:west authorsAll_txts_en:west`

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,authorsAll_txts_en&q=title_txt_en:west+authorsAll_txts_en:west'
```

Now let's say that we want to boost the documents where the author have the word "west" ahead of the documents where "west" was found in the title. To this we update the `q` parameter as follow `q=title_txt_en:west+authorsAll_txts_en:west^5` (notice the `^5` to boost the `authorsAll_txts_en` field)

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,authorsAll_txts_en&q=title_txt_en:west+authorsAll_txts_en:west^5'
```

Notice how documents where the author is named "West" come first, but we still get documents where the title includes the word "West".

If want to see why Solr ranked a result higher than another you can look at the `explain` information that Solr returns when passing the `debugQuery=on` parameter, for example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en,authorsAll_txts_en&q=title_txt_en:west+authorsAll_txts_en:west&debugQuery=on&wt=xml'
```

but be aware that the default `explain` output from Solr is rather convoluted. Take a look at this blog post to get primer on how to interpret this information.

Default Field (optional)

By default if you don't specify a field to search on the `q` parameter Solr will use a default field. In a typical Solr installation this would be the `_text_` field. For example if we issue a query for the word "west" without indicating a field (e.g. `q=west`) and look at the debug information we will see what Solr expanded the query into:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=west&debugQuery=on'

  # {
  #   "debug":{
  #     "rawquerystring":"west",
  #     "querystring":"west",
  #     "parsedquery":"_text_:west",
  #     "parsedquery_toString":"_text_:west",
  #     ...
  #   }
  # }
```

notice the `parsedquery` indicates that is searching on the `_text_` field.

You can overwrite the default field by passing the `df` parameter, for example to use the `title_txt_en` field as the default parameter we could pass `qf=title_txt_en:west`. This is somewhat similar to the Query Fields `qf` parameter that we saw before except that you can only indicate one `df` field. The advantage of `df` over `qf` is that `df` is supported by all Query Parsers whereas `qf` requires you to use DisMax or eDisMax.

Filtering with ranges

You can also filter a field to be within a range by using the bracket operator with the following syntax: `field:[firstValue TO lastValue]`. For example, to request documents with `id` between `00000018` and `00000028` we could do: `id:[00000018 TO 00000028]`. You can also indicate open-ended ranges by passing an asterisk as the value, for example: `id:[* TO 00000028]`.

Be aware that range filtering with `string` fields would work as you would expect it to, but with `text_general` and `text_en` fields it will filter on the *terms indexed* not on the value of the field.

Minimum match (optional)

In addition to using the `AND/OR` operators in our searches, the eDisMax parser provides a powerful feature called *minimum match* (`mm`) that allows for more flexible matching conditions than what we can do with just boolean operators.

```
The eDisMax query parser provides the ability to blur the lines of
traditional Boolean logic through the use of the mm (minimum match)
parameter. The mm parameter allows you to define either a specific
number of terms or a percentage of terms in a query that must match
in order for a document to be considered a match.
- [Solr in Action, p. 228]
```

With the *minimum match* parameter is possible to tell Solr to consider a document a match if 75% of the terms searched for are found on it for all queries that have more than three words. For example, the following four word query `q=school teachers secondary classroom` on the title field (`qf=title_txt_en`) will return any document where at least 50% of the search terms are found (`mm=3<50%`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?
defType=edismax&fl=id,title_txt_en&mm=3%3C50%25&q=school%20teachers%20secondary%20classroom&qf=title_txt_en'

  #
  # results will include
  #
  # {
  #   "id":"00010001",
  #   "title_txt_en":["Succeeding in the secondary classroom : strategies for middle and high school teachers
/"]},
  # {
  #   "id":"00002200",
  #   "title_txt_en":["Aids to teachers of School chemistry."]},
  # {
  #   "id":"00020157",
  #   "title_txt_en":["Standards in the classroom : how teachers and students negotiate learning /"]},
  # {
  #   "id":"00008378",
  #   "title_txt_en":["Keys to the classroom : a teacher's guide to the first month of school /"]},
  #
```

We can indicate more than one minimum match value in a single query. For example, we can indicate that if two words are entered in a query both of them are required, but if more than two words are entered we are OK if only 66% (2 out of 3 words) are found. The syntax for this kind of queries is a bit tricky, though: `mm=2<2&3<2`

Where to find more

Searching is a large topic and complex topic. I've found the book "Relevant search with applications for Solr and Elasticsearch" (see references) to be a good conceptual reference with specifics on how to understand and configure Solr to improve search results. Chapter 3 on this book goes into great detail on how to read and understand the ranking of results.

Facets

One of the most popular features of Solr is the concept of *facets*. The Solr Reference Guide defines it as:

```
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.
Faceting makes it easy for users to explore search results, narrowing
in on exactly the results they are looking for.
```

You can easily get facet information from a query by selecting what field (or fields) you want to use to generate the categories and the counts. The basic syntax is `facet=on` followed by `facet.field=name-of-field`. For example to facet our dataset by *subjects* we would use the following syntax: `facet.field=subjects_ss` as in the following example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*&facet=on&facet.field=subjects_ss'

  # result will include
  #
  # "facet_counts":{
  #  "facet_queries":{},
  #  "facet_fields":{
  #    "subjects_ss":[
  #      "Women",179,
  #      "African Americans",159,
  #      "Christian life",119,
  #      "Large type books",110,
  #      "Indians of North America",104,
  #      "English language",88,
  #      ...
  #
```

You might have noticed that we are using the `string` representation of the subjects (`subjects_ss`) to generate

the facets rather than the `text_en` version stored in the `subjects_txts_en` field. This is because, as the Solr Reference Guide indicates facets are calculated "based on indexed terms". The indexed version of the `subjects_txts_en` field is tokenized whereas the indexed version of `subjects_ss` is the entire string.

You can indicate more than one `facet.field` in a query to Solr (e.g. `facet.field=publisher_s&facet.field=subjects_ss`) to get facets for more than one field.

There are several extra parameters that you can pass to Solr to customize how many facets are returned on result set. For example, if you want to list only the top 20 subjects in the facets rather than all of them you can indicate this with the following syntax: `f.subjects_ss.facet.limit=20`. You can also filter only get facets that have *at least* certain number of matches, for example only subjects that have at least 50 books `f.subjects_ss.facet.mincount=50` as shown the following example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=*&facet=on&facet.field=subjects_ss&f.subjects_ss.facet.mincount=50&f.subjects_ss.facet.limit=20'
```

You can also facet **by multiple fields at once** this is called Pivot Faceting. The way to do this is via the `facet.pivot` parameter. This parameter allows you to list the fields that should be used to facet the data, for example to facet the information *by subject and then by publisher* you could issue the following command:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=*&facet=on&facet.pivot=subjects_ss,publisher_s&facet.limit=5'

  #
  # response will include facets organized as follows:
  #
  # "facet_counts":{
  #  "facet_pivot":{
  #    "subjects_str,publisher_str":[
  #      {
  #        "field":"subjects_str",
  #        "value":"Women",
  #        "count":179,
  #        "pivot":[
  #          { "field":"publisher_str", "value":"New York :", "count":24},
  #          { "field":"publisher_str", "value":"Berkeley Heights, NJ :", "count":19},
  #          { "field":"publisher_str", "value":"Minneapolis :", "count":11}
  #        ]
  #      },
  #      {
  #        "field":"subjects_str",
  #        "value":"African Americans",
  #        "count":159,
  #        "pivot":[
  #          { "field":"publisher_str", "value":"Berkeley Heights, NJ :", "count":35},
  #          { "field":"publisher_str", "value":"New York :", "count":22},
  #          { "field":"publisher_str", "value":"Chanhassen, MN :", "count":9}
  #        ]
  #      }
  #    ]
  #  ...
  #
```

Hit highlighting

Another Solr feature is the ability to return a fragment of the document where the match was found for a given search term. This is called highlighting.

Let's say that we search for books where the one of the authors (`authorsAll_txts_en`) or the title (`title_txt_en`) include the word "michael". If we add an extra parameter to the query `hl=on` to enable highlighting the results will include an indicator of what part of the author or the title has the match.

```
$ curl 'http://localhost:8983/solr/bibdata/select?
defType=edismax&q=michael&qf=title_txt_en+authorsAll_txts_en&hl=on'
```

```
#
# response will include
#
# "highlighting":{
#   "00008929":{
#     "title_txt_en":["<em>Michael</em> Jackson /"]},
#   "00022067":{
#     "authorsAll_txts_en":["Chinery, <em>Michael</em>."],
#     "title_txt_en":["Partners and parents / by <em>Michael</em> Chinery."]},
#   "00011434":{
#     "authorsAll_txts_en":["Castleman, <em>Michael</em>."]},
#
```

Notice how the `highlighting` property includes the `id` of each document in the result (e.g. `00008929`), the field where the match was found (e.g. `authorsAll_txts_en` and/or `title_txt_en`) and the text that matched within the field (e.g. `<em>Michael</em> Jackson /"`). You can display this information along with your search results to allow the user to "preview" why each result was rendered.

PART IV: MISCELLANEOUS (optional)

Solr directories

In the next sections we'll make a few changes to the configuration of our `bidata` core. Before we do that let's take a look at the files and directories that were created when we unzipped the `solr-7.4.0.zip` file.

Assuming we unzipped this zip file in our home directory we would have a folder `~/solr-7.4.0/` with several directories underneath:

```
~/solr-7.4.0/
|-- bin/
|-- dist/
|-- examples/
|-- server/
    |-- solr/
    |-- solr-webapp/
```

Directory `bin/` contains the scripts to start/stop Solr and post data to it.

Directory `dist/` contains the Java Archive (JAR) files. These are the binaries that make up Solr.

Directory `examples/` hold sample data that Solr provides out of the box. You should be able to import this data via the `post` tool like we did for our `bibdata` core.

Directory `server/solr/` contains one folder for each of the cores defined by default. For example, there should be a `bibdata` folder here for our core.

Directory `server/solr-webapp/` contains the code to power the "Solr Admin" that we see when we visit http://localhost:8983/solr/#/

Your bibdata core

As noted above, our `bibdata` core is under the `server/solr/bibdata` folder. The structure of this folder is as follows:

```
~/solr-7.4.0/
|-- server/
    |-- solr/
        |-- bibdata/
            |-- conf/
            |-- data/
```

The `data` folder contains the data that Solr stores for this core. This is where the actual index is located. The only thing that you probably want to do with this folder is back it up regularly. Other than that, you should stay away from it :)

The `conf` folder contains configuration files for this core. In the following sections we'll look at some of the files in this folder (e.g. `solrconfig.xml`, `managed-schema`, `stopwords.txt`, and `synonyms.txt`) and how they can be updated to configure different options in Solr.

Synonyms

In a previous section, when we looked at the `text_general` and `text_en` field types, we noticed that it used a filter to handle synonyms at query time.

Here is how to view that definition again:

```
$ curl 'http://localhost:8983/solr/bibdata/schema/fieldtypes/text_en'

  #
  # "queryAnalyzer":{
  #   "tokenizer":{
  #     ...
  #   },
  #   "filters":[
  #     ...
  #     {
  #       "class":"solr.SynonymGraphFilterFactory",
  #       "expand":"true",
  #       "ignoreCase":"true",
  #       "synonyms":"synonyms.txt"
  #     },
  #     ...
  #
```

Notice how one of the filter uses the `SynonymGraphFilterFactory` to handle synonyms and references a file `synonyms.txt`.

The file `synonyms.txt` can be found on the configuration folder for our `bibdata` core under `~/solr-7.4.0/server/solr/bibdata/conf/synonyms.txt` If you take a look at the contents of this file you'll see a definition for synonyms for "television"

```
$ cat ~/solr-7.4.0/server/solr/bibdata/conf/synonyms.txt

  #
  # will include a few lines including
  #
  # GB,gib,gigabyte,gigabytes
  # Television, Televisions, TV, TVs
  #
```

Life without synonyms

In the data in our `bibdata` core several of the books have the words "twentieth century" in the title but these books would not be retrieved if a user were to search for "20th century".

Let's try it, first let's search for `q=title_txt_en:"twentieth century"`:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:"twentieth+century"'

  #
  # result will include 29 results
  #
```

And now let's search for `q=title_txt_en:"20th century"`:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:"20th+century"'

  #
  # result will include 4 results
  #
```

Adding synonyms

We can indicate Solr that "twentieth" and "20th" are synonyms by updating the `synonyms.txt` file by adding a like as follows:

```
20th,twentieth
```

You can do this with your favorite editor or with a command like this:

```
$ echo "20th,twentieth" >> ~/solr-7.4.0/server/solr/bibdata/conf/synonyms.txt
```

You *must reload your core* for the changes to the `synonyms.txt` to take effect. You can do this as follow:

```
$ curl 'http://localhost:8983/solr/admin/cores?action=RELOAD&core=bibdata'

  # response will look similar to this
  # {
  # "responseHeader":{
  #   "status":0,
  #   "QTime":221}}
  #
```

You can also reload the core via the Solr Admin page. Select "Core Admin", then "bibdata", and click "Reload".

If you run the queries again they will both report "33 results found" regardless of whether you search for `q=title:"twentieth century"` or `q=title:"20th century"`:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:"twentieth+century"'

  #
  # result will include 33 results
  #
```

To find more about synonyms take a look at this blog post where I talk about the different ways of adding synonyms, how to test them in the Solr Admin tool, and the differences between applying synonyms at index time versus query time.

Core-specific configuration

One of the most important configuration files for a Solr core is `solrconfig.xml` located in the configuration folder for the core. In our `bibdata` core it would be located under `~/solr-7.4.0/server/solr/bibdata/conf/solrconfig.xml`.

A default `solrconfig.xml` file is about 1300 lines of heavily documented XML. We won't need to make changes to most of the content of this file, but there are a couple of areas that are worth knowing about: request handlers and search components.

Note: Despite its name, file `solrconfig.xml` controls the configuration *for our core*, not for the entire Solr installation. Each core has its own `solrconfig.xml` file. There is a separate file for Solr-wide configuration settings. In our Solr installation it will be under `~/solr-7.4.0/server/solr/solr.xml`. This file is out of the scope of this tutorial.

Request Handlers

When we submit a request to Solr the request is processed by a request handler. Throughout this tutorial all our queries to Solr have gone to a URL that ends with `/select`, for example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=*'
```

The `/select` in the URL points to a request handler defined in `solrconfig.xml`. If we look at the content of this file you'll notice a definition like this:

```
$ cat ~/solr-7.4.0/server/solr/bibdata/conf/solrconfig.xml

  #
  # notice the "/select" in this requestHandler definition
  #
```

```
  # <requestHandler name="/select" class="solr.SearchHandler">
  #   <lst name="defaults">
  #     <str name="echoParams">explicit</str>
  #     <int name="rows">10</int>
  #   </lst>
  # </requestHandler>
  #
```

We can make changes to this section to indicate that we want to use the eDisMax query parser (`defType`) by default and set the default query fields (`qf`) to title and author. To do so we could update the "defaults" section as follows:

```
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="defType">edismax</str>
    <str name="qf">title_txt_en authorsAll_txts_en</str>
  </lst>
```

We'll need to reload your core for changes to the `solrconfig.xml` to take effect.

Be careful, an incorrect setting on this file can take our core down or cause queries to give unexpected results. For example, entering the `qf` value as `title_txt_en, authorsAll_txts_en` (notice the comma to separate the fields) will cause Solr to ignore this parameter.

The Solr Reference Guide has excellent documentation on what the values for a request handler mean and how we can configure them.

LocalParams and dereferencing

In addition to the standard parameters in a request handler we can also define custom settings and use them in our search queries. For example is possible to define a new setting (`custom_search_field`) to group a list of fields and their boost values as shown below:

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    ...
  </lst>
  <str name="custom_search_field">
    title_txt_en^10
    authorsAll_txts_en^5
    subjectssubjects_txts_en
  </str>
</requestHandler>
```

We can then use this new setting in our queries by using the Local Parameters and Dereferencing features of Solr.

The syntax to use local parameters and dereferencing look a bit scary at first since you have to pass your parameters in the following format: `{! key=value}` where `key` is the parameter that you want to pass and `value` the value to use for that parameter. Dereferencing (asking Solr use a pre-existing value rather than a literal) is triggered by prefixing the `value` with a `$` as in `{! key=$value}`

For example to use our newly defined `custom_search_field` in a query we could pass the following to Solr:

```
q={! qf=$custom_search_field}teachers
```

You can see an example of how this is used in a Blacklight application in the following blog post.

Search Components

Request handlers in turn use search components to execute different operations on a search. The Solr Reference

Guide defines search components as:

```
A search component is a feature of search, such as highlighting or faceting.
The search component is defined in solrconfig.xml separate from the request
handlers, and then registered with a request handler as needed.
```

You can find the definition of the search components in the `solrconfig.xml` by looking at the `searchComponent` elements defined in this file. For example, in our `solrconfig.xml` there is a section like this for the highlighting feature that we used before:

```
<searchComponent class="solr.HighlightComponent" name="highlight">
  <highlighting>
    ... lots of other properties are define here...
    <formatter name="html"
      default="true"
      class="solr.highlight.HtmlFormatter">
      <lst name="defaults">
        <str name="hl.simple.pre"><![CDATA[<em>]]></str>
        <str name="hl.simple.post"><![CDATA[</em>]]></str>
      </lst>
    </formatter>
    ... lots of other properties are define here...
```

Notice that the HTML tokens (`<em>` and `</em>`) that we saw in the highlighting results in previous section are defined here.

Although search components are defined in `solrconfig.xml` it's a bit tricky to notice their relationship to request handlers in the config because Solr defines a set of default search components that are automatically applied *unless we overwrite them*.

Spellchecker

Solr provides spellcheck functionality out of the box that we can use to help users when they misspell a word in their queries. For example, if a user searches for "Washingon" (notice the missing "t") most likely Solr will return zero results, but with the spellcheck turned on Solr is able to suggest the correct spelling for the query (i.e. "Washington").

In our current `bibdata` core a search for "Washingon" will return zero results:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title&q=title:washingon'

  #
  # response will indicate
  # {
  # "responseHeader":{
  #   "status":0,
  #  "params":{
  #    "q":"title:washingon",
  #    "fl":"id,title"}},
  # "response":{"numFound":0,"start":0,"docs":[]
  # }}
  #
```

Spellchecking is configured under the `/select` request handler in `solrconfig.xml`. To enable it we need to update the `defaults` settings and enable the `spellcheck` search component. To do this update the `/select` request handler as follows:

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    <str name="echoParams">explicit</str>
    <int name="rows">10</int>
    <str name="defType">edismax</str>
    <str name="spellcheck">on</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>
  </lst>

  <arr name="last-components">
    <str>spellcheck</str>
  </arr>
</requestHandler>
```

The `spellcheck` component indicated above is already defined in the `solrconfig.xml` with the following defaults.

```
<searchComponent name="spellcheck" class="solr.SpellCheckComponent">
  <str name="queryAnalyzerFieldType">text_general</str>
  <lst name="spellchecker">
    <str name="name">default</str>
    <str name="field">_text_</str>
    <str name="classname">solr.DirectSolrSpellChecker</str>
    ...
  </lst>
</searchComponent
```

Notice how by default it will use the `_text_` field for spellcheck. The `_text_` field would be a good field to use if we were populating it, but we aren't in our current configuration. Instead let's update this setting to use the `title_txt_en` field instead.

Once these changes have been made to the `solrconfig.xml` we must reload our core for the changes to take effect:

```
$ curl 'http://localhost:8983/solr/admin/cores?action=RELOAD&core=bibdata'

  #
  # {
  #   "responseHeader":{
  #      "status":0,
  #      "QTime":10392
  #   }
  # }
  #
```

Now that our `bibdata` core has been configured to use spellcheck let's try out misspelled query again:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title_txt_en&q=title_txt_en:washingon'

  #
  # response will indicate
  #
  # {
  #   "responseHeader":{
  #     "response":{"numFound":0,"start":0,"docs":[]
  #   },
  #   "spellcheck":{
  #     "suggestions":[
  #       "washingon",{
  #         "numFound":1,
  #         "startOffset":6,
  #         "endOffset":15,
  #         "suggestion":["washington"]
  #     }],
  #     "collations":[
  #       "collation",{
  #         "collationQuery":"title_txt_en:washington",
  #         "hits":21,
  #         "misspellingsAndCorrections":[
  #         "washingon","washington"]}]
```

```
#   }
# }
#
```

Notice that even though we got zero results back, the response now includes a `spellcheck` section *with the words that were misspelled and the suggested spelling for it*. We can use this information to alert the user that perhaps they misspelled a word or perhaps re-submit the query with the correct spelling.

Sources and where to find more

- Solr Reference Guide
- Solr in Action by Trey Grainger and Timothy Potter
- Relevant search with applications for Solr and Elasticsearch by Doug Turnbull and John Berryman

Sample data

File `books.json` contains 10,000 books taken from Library of Congress' MARC Distribution Services.

The steps to create the `books.json` file from the MARC data are as follow:

- Download file `BooksAll.2014.part01.utf8.gz` from https://www.loc.gov/cds/downloads/MDSConnect/BooksAll.2014.part01.utf8.gz.
- Unzip it: `gzip -d BooksAll.2014.part01.utf8.gz`
- Process the unzipped file with marcli with the following command: `./marcli --file BooksAll.2014.part01.utf8 -format solr > books.json`

The MARC file has 250,000 books and therefore the resulting `books.json` will have 250,000 too. For the purposes of this tutorial I manually truncated the file to include only the first 10,000 books.

`marcli` is a small utility program that I wrote in Go to parse MARC files. If you are interested in the part that generates the JSON out of the MARC record take a look at the processorSolr.go file.