Solr for newbies workshop

Hector Correa
hector@hectorcorrea.com
http://hectorcorrea.com/solr-for-newbies

Table of Contents

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.

In earlier versions of Solr documents were self-contained and did not support nested documents. Starting with version 8 Solr provides support for nested documents. This tutorial does not cover nested documents.

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           Princeton guide for dog owners
2           Princeton tour guide
3           Cats and dogs
```

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

```
KEY         DOCUMENT ID
---------   -----------
princeton     1, 2
owners      1
dogs          1, 3
guide         1, 2
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 brain 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

To install Solr we are going to use a tool called Docker that allows us to download small virtual machines (called containers) with pre-installed software. In our case we'll download a container with Solr 9.1.0 installed on it and use that during the workshop.

NOTE: You can also download and install the Solr binaries directly on your machine without using Docker. You'll need to have the Java Development Kit (JDK) for this to method to work. If you are interested in this approach take a look at these instructions instead.

For the Docker installation let's start by going to https://www.docker.com/, download the "Docker Desktop", install it, and run it.

Once installed run the following command from the terminal to make sure it's running:

```
$ docker ps

  #
  # You'll see something like this
  # CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS
PORTS     NAMES
```

If Docker is *not* running we'll see an error that will indicate something along the lines of

```
Error response from daemon: dial unix docker.raw.sock:
connect: connection refused
```

If we see this error it could be that the Docker Desktop app has not fully started. Wait a few seconds and try again. We can also open the "Docker Desktop" app and see its status.

Creating a Solr container

Once Docker has been installed and it's up and running we can create a container to host Solr 9.1.0 with the following command:

```
$ docker run -d -p 8983:8983 --name solr-container solr:9.1.0

  #
  # You'll see something like this...
  #
  # Unable to find image 'solr:9.1.0' locally
  # 9.1.0: Pulling from library/solr
  # 846c0b181fff: Pull complete
  # ...
  # fc8f2125142b: Pull complete
  # Digest:
sha256:971cd7a5c682390f8b1541ef74a8fd64d56c6a36e5c0849f6b48210a47
2
  # Status: Downloaded newer image for solr:9.1.0
  #
47e8cd4d281db5a19e7bfc98ee02ca73e19af66e392e5d8d3532938af5a76e9
6
```

The parameter `-d` the previous command tells Docker to run the container in the background (i.e. detached) and the parameter `-p 8983:8983` tells Docker to forwards calls to *our* local port `8983` to the port `8983` on the container.

We can check that the new container is running with the following command:

```
$ docker ps -f name=solr-container

  #
  # You'll see something like this...
  #
  # CONTAINER ID   IMAGE        COMMAND
CREATED         STATUS         PORTS
NAMES
  # 47e8cd4d281d   solr:9.1.0   "docker-entrypoint.s…"   2
minutes ago   Up 2 minutes   0.0.0.0:8983->8983/tcp, :::8983-
>8983/tcp   solr-container
```

Notice that now we have a container NAMED `solr-container` using the IMAGE `solr:9.1.0`. We can check the status of Solr with the following command:

```
$ docker exec -it solr-container solr status

  # Found 1 Solr nodes:
  #
  # Solr process 15 running on port 8983
  # {
  #   "solr_home":"/var/solr/data",
  #   "version":"9.1.0
aa4f3d98ab19c201e7f3c74cd14c99174148616d - ishan - 2022-11-11
13:00:47",
  #   "startTime":"2023-01-12T20:48:46.084Z",
  #   "uptime":"0 days, 0 hours, 9 minutes, 15 seconds",
  #   "memory":"178.3 MB (%34.8) of 512 MB"}
```

We can also see Solr running by pointing your browser to http://localhost:8983/solr/ which will show the Solr Admin web page. In this page we can see that we do not have any cores defined to store data, we'll fix that in the next section. WARNING: Do not attempt to create a Solr cores via "Add Core" button in the Solr Admin page – that button only leads to pain.

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` with the following command:

```
$ docker exec -it solr-container solr create_core -c bibdata

  #
  # WARNING: Using _default configset with data driven schema
functionality. NOT RECOMMENDED for production use.
  #          To turn off: bin/solr config -c bibdata -p 8983
-action set-user-property -property update.autoCreateFields -
value false
  #
  # Created new core 'bibdata'
```

If we go back to http://localhost:8983/solr/ on our browser (we might need to refresh the page) we should see our newly created `bibdata` core available in the "Core Selector" dropdown list.

Now that our core has been created we can query it with the following command:

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

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

and we'll see `"numFound":0` indicating that there are no documents on it. We can also point our browser to http://localhost:8983/solr#bibdata/query and click the "Execute Query" button at the bottom of the page and see the same result.

Adding documents to Solr

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

```
$ curl -OL
https://raw.githubusercontent.com/hectorcorrea/solr-for-
newbies/main/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. We can take a look at it with something like `head books.json` or using the text editor of our choice. Below is an example on one of the books in this file:

```
{
  "id":"00008027",
  "author_txt_en":"Patent, Dorothy Hinshaw.",
  "authors_other_txts_en":["Muñoz, William,"],
  "title_txt_en":"Horses /",
  "responsibility_txt_en":"by Dorothy Hinshaw Patent ;
photographs by William
Muñoz.","publisher_place_str":"Minneapolis, Minn. :",
  "publisher_name_str":"Lerner Publications,",
  "publisher_date_str":"c2001.",
  "subjects_txts_en":["Horses","Horses"],
  "subjects_form_txts_en":["Juvenile literature"]
}
```

To import this data to our Solr we'll first *copy* the file to the Docker container

```
$ docker cp books.json solr-container:/opt/solr-
9.1.0/books.json
```

and then we *load it* to Solr:

```
$ docker exec -it solr-container post -c bibdata books.json

  #
  # /opt/java/openjdk/bin/java -classpath
/opt/solr/server/solr-webapp/webapp/WEB-INF/lib/solr-core-
9.1.0.jar ...
  # SimplePostTool version 5.0.0
  # Posting files to [base] url
http://localhost:8983/solr/bibdata/update...
  # 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: 0:00:01.951
```

Now if we re-run our query we should see some results:

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

  #
  # {
  #  "responseHeader":{
  #    "status":0,
  #    "QTime":0,
  #    "params":{
  #      "q":"*"}},
  #  "response":
{"numFound":30424,"start":0,"numFoundExact":true,"docs":[
  #      {
  #      ...the information for the first 10 documents will be
displayed here..
  #
```

Notice how the number of documents found is greater than zero (e.g. `"numFound":30424`)

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)
- **authors_other_txts_en**: list of other authors (MARC 700a)
- **title_txt_en**: title of the book (MARC 245ab)
- **publisher_name_str**: publisher name (MARC 260b)
- **subjects_txts_en**: an array of subjects (MARC 650a)

The suffix added to each field (e.g. `_txt_en`) is a hint for Solr to pick the appropriate field type for each field as it 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 give you a cryptic error message instead.

Try adding and removing some other fields to this list, for example, `fl=id,title_txt_en,author_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 the results 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?
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?
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?
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 "art history" using `q=title_txt_en:"art history"` (make sure the text "art history" is in quotes)

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

  # the results will include 6 documents with titles like:
  #
  #  "title":["... : strategies for middle and high school
teachers /"]},
  #  "title":["... Sunday school teachers ... /"]},
  #  "title":["... solutions for middle and high school
teachers /"]}]
  #
```

Notice how all three results have the term "art history" somewhere on the title. Now let's issue a slightly different query using `q=title_txt_en:"art history"~3` to indicate that we want the words "art" and "history" 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:"art+history"~3'
```

The result for this query will include a few more books (notice that `numFound` is now `10` instead of `6`) and some of the new tiles include

```
  #  "title_txt_en":"History of art /"},
  #  "title_txt_en":"American art : a cultural history /"},
  #  "title_txt_en":"The invention of art : a cultural history
/"},
  #  "title_txt_en":"A history of art in Africa /"}]
```

these new books include the words "art" and "history" but they don't have to be exactly next to each other, as long as they are close to each other they are considered a match (the `~3` in our query asks for "edit distance of 3").

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:"art history"` and not `q=title_txt_en:art history`. 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?
debugQuery=on&q=title_txt_en:"art+history"' | grep parsedquery

  #
  # "parsedquery":"PhraseQuery(title_txt_en:\"art histori\")",
  #
```

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?
debugQuery=on&q=title_txt_en:art+history' | grep parsedquery

  #
  # "parsedquery":"title_txt_en:art _text_:history",
  #
```

notice that Solr searched for the word "art" in the `title_txt_en` field but searched for the word "history" 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_txt_en:education&fl=id,title_txt_en'

  #
  # response will include
  #   "response":{"numFound":340,"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?
q=title_txt_en:education&facet=on&facet.field=subjects_txts_en_st
'


  # response will include something like this
  #
  # "facet_counts":{
  #   "facet_fields":{
  #     "subjects_txts_en_str":[
  #       "Education",58,
  #       "Educational change",16,
  #       "Multicultural education",15,
  #       "Education, Higher",14,
  #       "Education and state",13,
  # #
```

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 using the text version (`subjects_txts_en`) and notice the facets returned:

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

Updating documents

To update a document in Solr we have two options. The most common 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 `00007345` we would get:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=id:00007345'

  # "response":
{"numFound":1,"start":0,"numFoundExact":true,"docs":[
  # {
  #   "id":"00007345",
  #   "authorsOther_txts_en":["Giannakis, Georgios B."],
  #   "title_txt_en":"Signal processing advances in wireless
and mobile communications /",
  #   "responsibility_txt_en":"edited by G.B. Giannakis ...
[et al.].",
  #   "publisher_txt_en":"Upper Saddle River, NJ :",
  #   "subjects_txts_en":["Signal processing", "Wireless
communication systems"],
  #   "_version_":1755055519763005440}]
  # }}
  #
```

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. Notice that the command is issued against the `update` endpoint rather than the `select` endpoint we used in our previous commands.

```
$ curl -X POST --data '[{"id":"00007345","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 `00007345` again we will see the new data and notice that the fields that we did not provide during 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:00007345'

  # "response":{"numFound":1,"start":0,"docs":[
  # {
  #   "id":"00007345",
  #   "title_txt_en":"the new title",
  # }]}
  #
```

The second option to update a document in Solr is to via atomic updates in which we can indicate what fields of the document will be updated. Details of this method are out of scope for this tutorial but below is a very simple example to show the basic syntax, notice how we are using the `set` parameter in the `title_txt_en` field to indicate a different kind of update:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=id:00007450'
  #
  # "title_txt_en":"Principles of fluid mechanics /",
  #

$ curl -X POST --data '[{"id":"00007450","title_txt_en":
{"set":"the new title for 00007450"}}]'
'http://localhost:8983/solr/bibdata/update?commit=true'

$ curl 'http://localhost:8983/solr/bibdata/select?
q=id:00007450'
  #
  # title will say "the new title for 00007450"
  # and the rest of the fields will remain unchanged
  #
...
```

Deleting documents

To delete documents from the `bibdata` core we also use the `update` endpoint but the structure of the command is as follows:

```
$ curl 'http://localhost:8983/solr/bibdata/update?commit=true'
--data '<delete><query>id:00008056</query></delete>'
```

The body of the request (`--data`) indicates to Solr that we want to delete a specific document (notice the `id:00008056` 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 in the schema 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:

```
$ docker exec -it solr-container solr delete -c bibdata

  # Deleting core 'bibdata' using command:
  # http://localhost:8983/solr/admin/cores?
action=UNLOAD&core=bibdata&deleteIndex=true&deleteDataDir=true&de
e
```

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 similar 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, `description` based on the `text` field type, and `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 we 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 perform different transformation to the values as we ingest them. 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 at http://localhost:8983/solr/#/bibdata/schema 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 (rather long) response 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 (e.g. development vs production)

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 now 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.",
  "authors_other_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:

1. 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.)
2. If there is no exact match in the schema then Solr will look at the **dynamicFields** definitions to see if the field can be handled 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.
3. If no match is found in the dynamic fields either Solr will guess what is the best type to use based on the data for this field in the first document. This is what happened with the `subjects_txts_en` field (notice that this field ends with `_txts_en` rather than `_txt_en`). In this case, since there is no dynamic field definition to handle this ending, Solr guessed and created field `subjects_txts_en` as `text_general` (and additionally created a string version `subjects_txts_en_str` of the field). For production use Solr recommends to disable this automatic guessing, this is what the "WARNING: Using _default configset with data driven schema functionality. NOT RECOMMENDED for production use" was about when we first created our Solr core.

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.

Field: id

Let's look at the details of 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 the schema 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 our query 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.

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 Analysis Screen in the Solr Admin tool is a great way to see how a particular text is either indexed or queried by Solr *depending on the field type*. Point your browser to http://localhost:8983/solr/#/bibdata/analysis and try the following examples:

Here are 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. Naomi Dushay's CJK with Solr for Libraries is a great resource on this topic.

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 it's also possible to add a field that is neither stored nor indexed but that's beyond the scope of this tutorial.

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 because we don't expect to ever search by this field. Conversely, perhaps we want to create a field with a combination of values to optimize a particular kind of search, 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 (e.g. `_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 (because there was no definition for `_txts_en` fields).

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 would 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.

Delete the existing `bibdata` core in Solr

```
$ docker exec -it solr-container solr delete -c bibdata

  # Deleting core 'bibdata' using command:
  # http://localhost:8983/solr/admin/cores?
action=UNLOAD&core=bibdata...
```

Then re-create it

```
$ docker exec -it solr-container solr create_core -c bibdata

  # WARNING: Using _default configset with data driven schema
functionality.
  #          ...
  #
  # Created new core 'bibdata'
```

And finally query it (you should have zero documents since we have not re-imported the data)

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

  #
  # "response":{"numFound":0,"start":0,"docs":[]
  #
```

This time *before* we import the data in the `books.json` file we are going to add a few field definitions to the schema to make sure the data is indexed and stored 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 as `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 `authors_other_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 authors_other_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 `authors_all_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":[ "authors_all_txts_en" ]
    },
    {
      "source":"authors_other_txts_en",
      "dest":[ "authors_all_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. We'll skip that for now.

Customizing the subject field

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 this field is something that we got for free when Solr automatically guessed the field type to use for it (remember that Solr created an additional `subjects_txts_en_str` for us). Now that we are handling this field explicitly with our `*_txts_en dynamicField` definition we need to explicitly ask Solr to create this extra field for us:

```
$ 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
```

Populating the *text* field

As we saw earlier, by default, if no field is indicated in a search, Solr searches in the `_text_` field. This field is already defined in our schema but we are currently not populating it with anything since the field does not exist in our `books.json` data file. Let's fix that, let's tell Solr to copy the value of every field into the `_text_` field by using a `copyField` definition like the one below:

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

In a production environment we probably want to be a more selective on how we populate `_text_` but this will do for us.

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.

```
$ docker exec -it solr-container post -c bibdata books.json

# /opt/java/openjdk/bin/java -classpath ...
SimplePostTool version 5.0.0
Posting files to [base] url
http://localhost:8983/solr/bibdata/update...
Entering auto mode. ...
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: 0:00:02.871
```

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 practical guide to creating and
maintaining water quality /",
  #  "title_txt_en":"A practical guide to particle counting
for drinking water treatment /",
  #  "title_txt_en":"Applied ground-water hydrology and well
hydraulics /",
  #  "title_txt_en":"Assessment of blue-green algal toxins in
raw and finished drinking water /",
  #  "title_txt_en":"Bureau of Reclamation..."
  #  "title_txt_en":"Carry me across the water : a novel /",
  #  "title_txt_en":"Clean Water Act : proposed revisions to
EPA regulations to clean up polluted waters /",
  #  "title_txt_en":"Cold water burning /"
  #  ...
  #
```

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 `authors_all_txts_en` field has the combination of `author_txt_en` and `authors_other_txts_en` even though our source data didn't have an `authors_all_txts_en` field:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=id:00009214'

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

Likewise, let's search for books authored by "Gallop" using our new `authors_all_txts_en` field (`q=authors_all_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.

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

Testing the *text* field

Let's run a query *without* specifing what field to search on, for example `q:biology`

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=biology&debug=all'
```

The result will include all documents where the word "biology" is found in the `_text_` field and since we are now populating this field with a copy of every value in our documents this means that we'll get back any document that has the word "biology" in the title, the author, or the subject.

We can confirm that Solr is searching on the `_text_` field by looking at the information in the parsed query, it will looks like this:

```
  "debug":{
    "rawquerystring":"biology",
    "querystring":"biology",
    "parsedquery":"_text_:biology",
```

Notice that our raw query `"biology"` got parsed as `"_text_:biology"`.

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:"art history"~3` when we wanted to search for books with the words "art" and "history" 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) "supports a robust and fairly intuitive syntax allowing you to create a variety of structured queries. The largest disadvantage is that it's very intolerant of syntax errors, as compared with something like the DisMax Query Parser which is designed to throw as few errors as possible."
- The DisMax query parser (DisMax) interface "is more like that of Google than the interface of the 'lucene' Solr query parser. This similarity makes DisMax the appropriate query parser for many consumer applications. It accepts a simple syntax, and it rarely produces error messages."
- The Extended DisMax (eDisMax) query parser is an improved version of the DisMax parser that is also very forgiving on 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 `curl` 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=authors_other_txts_en:*`), the `*` means "any value".

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

- Documents that do *not* have additional authors (`q=NOT authors_other_txts_en:*`) be aware that the `NOT` **must be in uppercase**.

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

- 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_txt_en,subjects_txts_en&q=subjects_txts_en:com*'
```

- Documents where title include "science" *and* at least one of the subjects is "women" (`q=title_txt_en:science AND subjects_txts_en:women` notice that both search conditions are indicated in the `q` parameter) Again, 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:science+AND+su
'
```

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

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title_txt_en&q=title_txt_en:history+AND+NOT+title_txt_en:ar
'
```

The Solr Reference Guide and this Lucene tutorial are good places to check for

quick reference on the query syntax.

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.

If we want to search for the same value in multiple fields at once (e.g. if we want to find all books where *the title or the author* includes the text "Washington") we must indicate each field/value pair individually: `q=title_txt_en:"Washington" authors_all_txts_en:"Washington"`.

The `qf` parameter allows us specify the fields separate from the terms so that we can use instead: `q="Washington"` and `qf=title_txt_en authors_all_txts_en`. This is really handy if we want to customize what fields are searched in an application in which the user enters a single text (say "Washington") and the application automatically searches multiple fields.

Below is an example of this (remember to select the eDisMax parser (`defType=edismax`) when using the `qf` parameter):

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q="washington"&qf=title_txt_en+authors_all_txts_en&defType=edisma
'
```

debugQuery

Solr provides an extra parameter `debug=all` that we can use to get debug information about a query. This is particularly useful if the results that we get are not what we were expecting. For example, let's run the same query again but this time passing the `debug=all` parameter:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q="washington"&qf=title_txt_en+authors_all_txts_en&defType=edisma
'


  # response will include
  # {
  #   "responseHeader":{...}
  #   "response":{...}
  #   "debug":{
  #     "rawquerystring":"\"washington\"",
  #     "querystring":"\"washington\"",
  #
"parsedquery":"+DisjunctionMaxQuery((title_txt_en:washington |
authors_all_txts_en:washington))",
  #     "parsedquery_toString":"+(title_txt_en:washington |
authors_all_txts_en:washington)",
  #     "explain":{
  #       ... tons of information here ...
  #     }
  #     "QParser":"ExtendedDismaxQParser",
  #   }
  # }
  #
```

Notice the `debug` property in the output, 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 server
- 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 `art history` without surrounding them in quotes and the parsed query showed that it was querying two different fields `title_txt_en` for "art" and `_text_` for "history")
- you can also see that some of the search terms were stemmed (e.g. if you query for "running" you'll notice that the parsed query will show "run")
- how each document was ranked (`explain`)
- what query parser (`QParser`) was used

Check out this blog post for more information about `debugQuery`.

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 considers this when ranking documents that match a given query.

Let's say that we want documents where the word "Washington" (`q=washington`) is found in the title or in the author (`qf=title_txt_en authors_all_txts_en`)

```
$ curl 'http://localhost:8983/solr/bibdata/select?
&q=washington&qf=title_txt_en+authors_all_txts_en&defType=edismax
'
```

Now let's say that we want to boost the documents where the author has the word "Washington" ahead of the documents where "Washington" was found in the title. To this we update the `qf` parameter as follows `qf=title_txt_en authors_all_txts_en^5` (notice the `^5` to boost the `authors_all_txts_en` field)

```
$ curl 'http://localhost:8983/solr/bibdata/select?
&q=washington&qf=title_txt_en+authors_all_txts_en^5&defType=edism
'
```

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

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 authors_all_txts_en^-10`). They are just a way for us to hint Solr which fields we consider more important in a particular search.

If want to see why Solr ranked a result higher than another you can pass an additional parameter `debug.explain.structured=true` to see the explanation on how Solr ranked each of the documents in the result:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=title_txt_en:west+authors_all_txts_en:washington&debug=all&debu
'
```

The result will include an `explain` node with a ton of information for each of the documents ranked. This information is rather complex but it has a wealth of details that could help us figure out why a particular document is ranked higher or lower than what we would expect. Take a look at this blog post to get an idea on how to interpret this information.

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 `00010500` and `00012050` we could do: `id:[00010500 TO 00012050]`. You can also indicate open-ended ranges by passing an asterisk as the value, for example: `id:[* TO 00012050]`.

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=id:\
[00010500+TO+00012050\]'
```

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.

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",435,
  #      "Large type books",415,
  #      "African Americans",337,
  #      "English language",330,
  #      "World War, 1939-1945",196,
  #      ...
  #
```

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.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.

Note: Unfortunately the `facet.pivot` parameter is not available via the Solr Admin web page, if you want to try this example you will have to do it via the

command on the terminal.

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* (`facet.pivot=subjects_ss,publisher_name_str`) you could issue the following command:

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


  #
  # response will include facets organized as follows:
  #
  # "facet_counts":{
  #   "facet_pivot":{
  #     "subjects_ss,publisher_name_str":[{
  #     "field":"subjects_ss",
  #       "value":"Women",
  #         "count":435,
  #         "pivot":[{
  #           "field":"publisher_name_str",
  #           "value":"Chelsea House Publishers,",
  #           "count":22},
  #           {
  #           "field":"publisher_name_str",
  #           "value":"Enslow Publishers,",
  #           "count":13},
  #           ...
  #        ]
  #      }
  #    ]
  #  ...
  #
```

Notice how the results for the subject "Women" (435 results) are broken down by publisher under the "pivot" section.

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

(`authors_all_txts_en`) or the title (`title_txt_en`) include the word “Washington”. 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=washington&qf=title_txt_en+authors_all_txts_en&
'


  #
  # response will include a highlight section like this
  #
  # "highlighting":{
  #   "00065343":{
  #     "title_txt_en":["<em>Washington</em> Irving's The
legend of Sleepy Hollow.."],
  #     "authors_all_txts_en":["Irving,
<em>Washington</em>,"]},
  #   "00107795":{
  #     "authors_all_txts_en":["<em>Washington</em>,
Durthy."]},
  #   "00044606":{
  #     "title_txt_en":["University of <em>Washington</em>
/"]},
  #
```

Notice how the `highlighting` property includes the `id` of each document in the result (e.g. `00065343`), the field where the match was found (e.g. `authors_all_txts_en` and/or `title_txt_en`) and the text that matched within the field (e.g. `University of <em>Washington</em> /`). 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)

In the next sections we'll make a few changes to the configuration of our `bidata` core in order to enable some other features of Solr like synonyms and spell checking.

Solr's directories and configuration files

In Linux, Solr is typically installed under the `/opt/solr` folder and the data for our cores is stored under the `/var/solr/data` folder. We can see this in our Docker container if we log into it.

Open a separate terminal window and execute the following command to login into the container and see the files inside it:

```
$ docker exec -it solr-container /bin/bash
$ ls -la

  #
  # You'll see something like this
  #
  # bin     CHANGES.txt  docker   lib       LICENSE.txt
NOTICE.txt       README.txt
  # books.json  contrib  example  licenses  modules
prometheus-exporter  server
  #
```

While still on the Docker container issue a command as follow to see the files with the configuration for our `bibdata` core:

```
$ ls -la /var/solr/data/bibdata/conf/

  #
  # You'll see something like this
  #
  # drwxr-xr-x 2 solr solr     4096 Nov 11 07:31 lang
  # -rw-r--r-- 1 solr solr    26665 Jan 15 18:07 managed-
schema.xml
  # -rw-r--r-- 1 solr solr      873 Nov 11 07:31 protwords.txt
  # -rw-r--r-- 1  503 dialout 48192 Jan 15 19:45
solrconfig.xml
  # -rw-r--r-- 1 solr solr      781 Nov 11 07:31 stopwords.txt
  # -rw-r--r-- 1 solr solr     1124 Nov 11 07:31 synonyms.txt
```

Notice the `solrconfig.xml`, `managed-schema.xml` and the `synonyms.txt` files. These are the files that we saw before under the "Files" option in the Solr Admin web page.

File `managed-schema.xml` is where field definitions are declared. File `solrconfig.xml` is where we configure many of the features of Solr for our particular `bibdata` core. File `synonyms.txt` is where we define what words are considered synonyms and we'll look closely into this next.

Before we continue let's exit from the Docker container with the `exit` command (don't worry the Docker container is still up and running in the background):

```
$ exit
```

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":[
  #     ...
  #     a few filters go here
  #     ...
  #     {
  #       "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`.

You can view the contents of the `synonyms.txt` file for our `bibdata` core through the Files option in the Solr Admin web page:
http://localhost:8983/solr/#/bibdata/files?file=synonyms.txt

The contents of this file looks more or less like this:

```
  # Some synonym groups specific to this example
  GB,gib,gigabyte,gigabytes
  MB,mib,megabyte,megabytes
  Television, Televisions, TV, TVs
  #notice we use "gib" instead of "GiB" so any
WordDelimiterGraphFilter coming
  #after us won't split it into two words.

  # Synonym mappings can be used for spelling correction too
  pixima => pixma
```

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 84 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 22 results
  #
```

Adding synonyms

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

```
20th,twentieth
```

Because our Solr is running inside a Docker container we need to update the `synonyms.txt` file *inside* the container. We are going to do this in four steps:

1. First we'll copy `synonyms.txt` from the Docker container to our machine
2. Then we'll update the file in our machine (with whatever editor we are comfortable with)
3. Next we'll copy our updated local copy back to the container
4. And lastly, we'll tell Solr to reload the core's configuration so the changes take effect.

To copy the `synonyms.txt` from the container to our machine we'll issue the following command:

```
$ docker cp solr-
container:/var/solr/data/bibdata/conf/synonyms.txt .
$ ls

  #
  # drwxr-xr-x   3 user-id  staff    96 Jan 16 18:02 .
  # drwxr-xr-x  51 user-id  staff  1632 Jan 12 20:10 ..
  # -rw-r--r--@  1 user-id  staff  1124 Nov 11 02:31
synonyms.txt
  #
```

We can view the contents of the file with a command as follows:

```
$ cat synonyms.txt

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

Let's edit this file with whatever editor your are comfortable. Our goal is to add a new line to make `20th` and `twentieth` synonyms, we can do it like this:

```
$ echo "20th,twentieth" >> synonyms.txt
```

Now that we have updated our local copy of the synonyms file we need to copy this new version back to the Docker container, we can do this with a command like this:

```
$ docker cp synonyms.txt solr-
container:/var/solr/data/bibdata/conf/
```

If we refresh the page http://localhost:8983/solr/#/bibdata/files?file=synonyms.txt on our browser we should see that the new line has been added to the `synonyms.txt` file. However, we *must reload our core* for the changes to take effect. We 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 "106 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 106 results
  # 88 with "twentieth century" plus 22 with "20th century"
  #
```

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. We can view the content of this file in our `bibdata` core in this URL http://localhost:8983/solr/#/bibdata/files?file=solrconfig.xml

A default `solrconfig.xml` file is about 1100 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.

To make things easier for the rest of this section let's download two copies of this file to our local machine:

```
$ docker cp solr-
container:/var/solr/data/bibdata/conf/solrconfig.xml
solrconfig.xml
$ docker cp solr-
container:/var/solr/data/bibdata/conf/solrconfig.xml
solrconfig.bak
$ ls

  #
  # drwxr-xr-x   4 user-id  staff    128 Jan 16 18:19 .
  # drwxr-xr-x  51 user-id  staff   1632 Jan 12 20:10 ..
  # -rw-r--r--@  1 user-id  staff  47746 Jan 16 18:36
solrconfig.bak
  # -rw-r--r--@  1 user-id  staff  47746 Jan 16 18:36
solrconfig.xml
  # -rw-r--r--@  1 user-id  staff   1151 Jan 16 18:12
synonyms.txt
  #
```

`solrconfig.xml` is the file that we will be working on. Like with the `synonyms.txt` file before, the general workflow will be to make changes to this local version of the file, copy the updated file to the Docker container, and reload the Solr core to pick up the changes.

`solrconfig.bak` on the other hand is just backup, in case we mess up `solrconfig.xml` and need to go back to a well-known state.

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 (around like 733) a definition that looks like the one below, notice the `"/select"` in this request handler definition:

```
  #
  # <!-- Primary search handler, expected by most clients,
examples and UI frameworks -->
  # <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:

```
  <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="qf">title_txt_en authors_all_txts_en</str>
    </lst>
  </requestHandler>
```

We need to copy our updated file back to the Docker container and reload the core for the changes to take effect, let's do this with the following commands:

```
$ docker cp solrconfig.xml solr-
container:/var/solr/data/bibdata/conf/
$ curl 'http://localhost:8983/solr/admin/cores?
action=RELOAD&core=bibdata'
```

notice now now we we can issue a much simpler query since we don't have to specify the `qf` or `defType` parameters in the URL:

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

Be careful, an incorrect setting on the `solrconfig.xml` file can take our core down or cause queries to give unexpected results. For example, entering the `qf` value as `title_txt_en, authors_all_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.

Search Components

Request handlers in turn use *search components* to execute different operations on a search. Solr comes with several built-in default search components to implement faceting, highlighting, and spell checking to name a few.

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.

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_txt_en&q=title_txt_en: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 let's edit our local `solrconfig.xml` and replace the `<requestHandler name="/select" class="solr.SearchHandler">` node again but now with the following content:

```
<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>
```

and copy our updated version back to the Docker container and reload it:

```
$ docker cp solrconfig.xml solr-
container:/var/solr/data/bibdata/conf/
$ curl 'http://localhost:8983/solr/admin/cores?
action=RELOAD&core=bibdata'
```

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.

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

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

  #
  # response will still indicate zero documents found,
  # but the spellcheck node will be populated
  #
  # "response":
{"numFound":0,"start":0,"numFoundExact":true,"docs":[]},
  # "spellcheck":{
  #   "suggestions":[
  #     "washingon",{
  #       "numFound":3,
  #       "startOffset":13,
  #       "endOffset":22,
  #       "suggestion":["washington",
  #         "washigton",
  #         "washing"]}],
  #   "collations":[
  #     "collation",{
  #       "collationQuery":"title_txt_en:washington",
  #       "hits":41,
  #       "misspellingsAndCorrections":[
  #         "washingon","washington"]},
```

Notice that even though we still got zero results back (`numFound:0`), 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.

REFERENCES

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 roughly 30,000 books taken from the Library of Congress' MARC Distribution Services.

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

- Download file `BooksAll.2016.part01.utf8.gz` from https://www.loc.gov/cds/downloads/MDSConnect/BooksAll.2016.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.2016.part01.utf8 -match 2001 -matchFields 260 -format solr > books.json` to include only books published in 2001. The original MARC file has 250,000 books but the resulting file only includes 30,424 records.

`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.

Acknowledgements

I would like to thank my former team at the Brown University Library for their support and recommendations as I prepared the initial version of this tutorial back in 2017 as well as those that attended the workshop at the Code4Lib conference in Washington, DC in 2018 and San Jose, CA in 2019. A special thanks goes to Birkin Diana for helping me run the workshop in 2018 and 2019 and for taking the time to review the materials (multiple times!) and painstakingly testing each of the examples.