Solr for newbies workshop

Hector Correa
Senior Library Applications Developer
Brown University
hector_correa@brown.edu
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.

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 "9.0.1"
  # Java(TM) SE Runtime Environment (build 9.0.1+11)
  # Java HotSpot(TM) 64-Bit Server VM (build 9.0.1+11, mixed mode)
  #
```

If the JDK is installed on your machine you'll see the text indicating the version that you have (e.g. "9.0.1" 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 select the option to download the "Java Platform (JDK) 9".

From there, under the "Java SE Development Kit 9.0.1" select the file appropriated for your operating system, accept the license agreement, and download it. For example, for the Mac the file would be `jdk-9.0.1_osx-x64_bin.dmg`.

Run the installer that you downloaded. Once it has completed, go back to the Terminal 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://mirror.cc.columbia.edu/pub/software/apache/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

In the last section we ran a query against Solr that showed us that our newly created core `bibdata` has no documents in it. Remember that our call to

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

returned `"numFound":0`. Now let's add a few documents to this `bibdata` core. First, download this sample data file (if you cloned this GitHub repo the file is already in your machine):

```
$ 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. Go ahead and take a look at it (e.g. via `cat books.json`)

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

```
$ ~/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 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**: string for the main author (MARC 100a)
- **authorDate**: date for the author (MARC 100d)
- **authorFuller**: fuller form of the name (MARC 100q)
- **authorsOther**: list of other authors (MARC 700a)
- **title**: title of the book (MARC 245ab)
- **responsibility**: statement of responsibility (MARC 245c)
- **publisher**: publisher name (MARC 260a)
- **urls_ss**: URLs (MARC 856u)
- **subjects**: an array of subjects (MARC 650a)
- **subjectsForm**: (MARC 650v)
- **subjectsGeneral**: (MARC 650x)
- **subjectsChrono**: (MARC 650y)
- **subjectsGeo**: (MARC 650z)

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` of the documents we would use `fl=id,title` as in the following example:

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

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` and not `fl=id, title`. If the parameter includes spaces Solr will not return any results and it won't give you a visible error either!

Try adding and removing some other fields to this list, for example, `fl=id,author,title` or `fl=id,title,author,subjects`

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` has the word "teachers" we would use `q=title:teachers`:

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

We can request filter by many different fields, for example to request documents where the `title` includes the word "teachers" **or** the `author` includes the word "Alice" we would use `q=title:teachers author:Alice`

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title,author&q=title:teachers+author: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:teachers AND author:Alice` Please notice that the `AND` operator **must be in uppercase**.

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

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

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title&q=title:"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:"school teachers"~3` to indicate that we want the words "school" and "teachers" to be present in the `title` but they can be a few words apart (notice the `~3`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title&q=title:"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:"school teachers"` and not `q=title:school teachers`. The later will execute a search for "school" in the `title` 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 'http://localhost:8983/solr/bibdata/select?
fl=id,title&debugQuery=on&q=title:"school+teachers"' | grep parsedquery

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

whereas in the following command we don't:

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

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

Notice that Solr returns results paginated, by default it returns the first 10 documents that match the query. We can request a large

page size or another page via the `start` and `rows` parameters which we will discuss later. But 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:education`) and retrieve facets (`facet=on`) based on the `subjects` field (`facet.field=subjects`) we’ll use a query like this:

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

  # response will include something like this
  #
  # "facet_counts":{
  #   "facet_queries":{},
  #   "facet_fields":{
  #     "subjects":[
  #       "education",45,
  #       "and",18,
  #       "higher",12,
  #       "colleges",7,
  #       "in",7,
  #       "universities",7,
  #       "educational",6,
  #       "moral",4,
  #       "social",4,
  #       "teachers",4,
  #
```

You might notice a few extraneous subjects in the list (like the words “and”, and “in”) and you can also guess that “higher” should really be “higher education”. We’ll review later why we are getting the *tokenized* version of the subject rather than the actual subject values. For now, try issuing the previous command but using the `subjects_str` field instead:

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

  # response will include something like this
  #
  # "facet_counts":{
  #   "facet_queries":{},
  #   "facet_fields":{
  #     "subjects_str":[
  #       "Education, Higher",10,
  #       "Education",7,
  #       "Universities and colleges",6,
  #       "Educational equalization",3,
  #       "Moral education",3,
  #       "Women",3,
```

```
  #
```

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: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:

```
$ ~/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`, and `text_general`. 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 of 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, you can view how the details of it via the Schema API as shown in the following example. The response will be rather long but it will be roughly include the following categories under the "schema" element:

```
$ 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]
  #   }
  # }
  #
```

You can also view this information via the Solr Admin web page, under the *Schema* menu option or by looking at the `managed-schema` file under the *Files* menu option.

You can request each of these categories individually 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 new 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 where did the fields like `id`, `title`, `author`, `subjects`, and `subjects_str` in our `bibdata` core come

from since 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 some of the fields defined in our schema.

You can disable the automatic creation of fields in Solr if you don't want this behavior. Keep in mind that if automatic field creation is disabled Solr will *reject* the import of any documents with fields not defined in the schema. Note: I believe the ability to disable automatic field creation is new in Solr 6.x. Need to find out the exact version this became available.

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 that will be used to handle values of the string type.

Field: title

Now let's look at a more complex field and field type. Let's look at the definition for the `title` type:

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

  # {
  #   "responseHeader":{...}
  #   "field":{
  #     "name":"title",
  #     "type":"text_general"
```

```
  #   }
  # }
  #
```

That looks innocent enough: our `title` field points to the type `text_general` so let's see what does `text_general` means:

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

  # {
  #   "responseHeader":{...}
  #   "fieldType":{
  #     "name":"text_general",
  #     "class":"solr.TextField",
  #     "positionIncrementGap":"100",
  #     "multiValued":true,
  #     "indexAnalyzer":{
  #       "tokenizer":{
  #         "class":"solr.StandardTokenizerFactory"
  #       },
  #       "filters":[
  #         {
  #           "class":"solr.StopFilterFactory",
  #           "words":"stopwords.txt",
  #           "ignoreCase":"true"
  #         },
  #         {
  #           "class":"solr.LowerCaseFilterFactory"
  #         }
  #       ]
  #     },
  #     "queryAnalyzer":{
  #       "tokenizer":{
  #         "class":"solr.StandardTokenizerFactory"
  #       },
  #       "filters":[
  #         {
  #           "class":"solr.StopFilterFactory",
  #           "words":"stopwords.txt",
  #           "ignoreCase":"true"
  #         },
  #         {
  #           "class":"solr.SynonymGraphFilterFactory",
  #           "expand":"true",
  #           "ignoreCase":"true",
  #           "synonyms":"synonyms.txt"
  #         },
  #         {
  #           "class":"solr.LowerCaseFilterFactory"
  #         }
  #       ]
  #     }
  #   }
  # }
```

This is obviously a much more complex definition than the ones we saw before. Although the basics are the same, the field type points to a `class` (solr.TextField) and it indicates it's a multi-value type, notice the next two properties defined for this field: `indexAnalyzer` and `queryAnalyzer`. We will explore those in the next section.

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 the `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 the `tokenizer` and then by the `filters` defined in the `queryAnalyzer` section for that field.

If we look again at the definition for the `text_general` 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 `SynonymGraphFilter` 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

If we look at the tokenizer and filters defined for the `text_general` field type at *index time* we would see that we are tokenizing with the `StandardTokenizer` and filtering with the `StopFilter` and the `LowerCaseFilter` filters.

Notice that the at *query time* an extra filter, the `SynonymGraphFilter`, is applied for this field type.

If we *index* the text “The television is broken!” the tokenizer and filters defined in the `indexAnalyzer` will transform this text to four tokens: “the”, “television”, “is”, and “broken”. Notice how the tokens were lowercased (“The” became “the”) and the exclamation sign was dropped.

Likewise, if we *query* for the text “The TV is broken!” the tokenizer and filters defined in the `queryAnalyzer` will convert the text to the following tokens: “the”, “television”, “televisions”, “tvs”, “tv”, “is”, and “broken”. Notice that an additional transformation was done to this text, namely, the word “TV” was expanded to four synonyms. This is because the `queryAnalyzer` uses the `SynonymGraphFilter` and a standard Solr configuration comes with those four 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 see how is indexed. Lastly, select `text_en` and see how is indexed.
- 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”. Try again with the different field types to see the matches.
- Now enter “The TV is broken!” on the “Field Value (*index*)” text box, blank the “Field Value (*query*)” text box, select `text_general`, and see how the value is indexed. Then do the reverse, blank 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_general` fields.

Quiz: When we tested the text “The television is broken” with the `text_general` field type we probably expected the word “the” to be dropped since it’s a stop word in the English language. Can you guess why it was not dropped? Hint: try the same text but with the `text_en` field type instead and see what happens.

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 several filters (`CJKWidthFilterFactory`, `LowerCaseFilterFactory`, `CJKBigramFilterFactory`) that are different from what we saw in the `text_general` 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 and enter “胡志明” (Ho Chi Minh) as the “Field Value (index)” and the “Field Value (Query)”, select `text_en` as the FieldType and analyse the values you’ll notice how there is (as expected) a match. Now change the “Field Value (Query)” to include only the “志” character and run the analysis again using `text_en` as the field type. Notice how Solr will detect a match, which is incorrect using CJK rules.

If you select `text_cjk` as the FieldType and run the analysis again you'll see how Solr detected that this was not a match after applying the rules of the `CJKBigramFilterFactory`.

The data for this section was taken from this blog post. Although the blog post is a bit dated, there basic idea of it is 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.

But why?

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, although we are not populating it).

Indexed, stored, and docValues

In the previous example we declared three fields with different indexed/stored settings and all of them were of type `text_general`. However, if you set the indexed/stored setting in fields of type `string` the default behavior is slightly different because Solr stores and handles `string` fields different from `text_general` fields.

For `string` fields, Solr adds an extra property DocValues and sets it to true by default. This is an optimization that allows Solr to perform better when searching and faceting values in these fields but, as a result, fields with `docValue=true` behave as fields *indexed and stored* regardless of the value set for `stored` and `indexed` properties!

You can prevent Solr from using `docValues` in a `string` field by setting `docValues=false`. This would allow you to control whether a field is really stored and index like we did for `text_general` fields. Unless you have a good reason to change the default value for `docValues` I would suggest not changing it.

Solr does not allow you to set `docValues=true` for `text_general` fields.

Recreating our Solr core

Before we start the next section, where we will make customizations to the schema, let's delete the current core and re-create it empty.

To delete the `bibdata` core issue:

```
$ solr delete -c bibdata

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

To re-create the core issue:

```
$ solr create -c bibdata

  #
  # Created new core 'bibdata'
  #
```

Making sure the core was created:

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

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

Adding a new field

So far we have only worked with the fields that were automatically added to our `bibdata` core as we imported the data. Let us now add and customize some of the fields in our core to have more control on how Solr indexes and searches data.

Customizing the author fields

Our JSON file with the source data has a main author (the `author` property) and other authors (the `authorsOther` property). We know `author` is single value and `authorsOther` is multi-value. When we let Solr automatically create these fields both of them ended up being multi-value so let's define them manually in our schema so that we can customize author to be single value so that we can sort our results by it (you can only sort by single-value fields).

Let's also create a new field (`authorsAll`) that will combine the main author with the other authors, that way we only need to search against a single field when searching by author. Notice that we are making this field `text_general` (rather than string) so that we can do partial matching.

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":[
    {
      "name":"author",
      "type":"string",
      "multiValued":false
    },
```

```
    {
      "name":"authorsOther",
      "type":"string",
      "multiValued":true
    },
    {
      "name":"authorsAll",
      "type":"text_general",
      "multiValued":true
    }
  ]
}' http://localhost:8983/solr/bibdata/schema

  #
  # response will look like
  # {
  #   "responseHeader":{
  #     "status":0,
  #     "QTime":10}}
  #
```

Now let's configure Solr to automatically copy the values of `author` and `authorOther` to our new `authorAll` field by defining two `copy-fields`:

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

We can see how these changes are now part of the schema by looking at the definitions of the `author`, `authorsOther`, and `authorsAll` in the Solr Admin tool either via the *Schema* menu option or by looking at the `managed-schema` file under the *Files* option.

We need to re-import our data for these changes to be applied to the data, but before we do this let's do another customization to the schema.

Customizing the title fields

Most (if not all) the titles in our source JSON file are in English. Therefore let's configure the `title` field to be single-value and to use the `text_en` (text English) field type rather than the default multi-value `text_general` field type.

Field type `text_general` uses the standard tokenizer and two basic filters (StopFilter and LowerCase). In contrast `text_en` uses a similar configuration but it adds three more filters to the definition (EnglishPossessive, KeywordMarker, and PorterStem) that allow for more sophisticated queries. We can run `curl localhost:8983/solr/bibdata/schema/fieldtypes/text_general` and `curl localhost:8983/solr/bibdata/schema/fieldtypes/text_en` to see the specifics for these field types.

When we let Solr automatically create the `title` field based on the data that we imported, Solr also created a second field (`title_str`) with the string representation of the title. Now that we are explicitly defining the `title` field Solr won't automatically create the second field for us, but we can easily ask Solr to do it. We'll use the `_s` instead of `_str` because our new title field is single value.

```
$ curl -X POST -H 'Content-type:application/json' --data-binary '{
  "add-field":{
```

```
    "name":"title",
    "type":"text_en",
    "multiValued":false
  },
  "add-copy-field":{
    "source":"title",
    "dest":[ "title_s" ]
  }
}' 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 author field

Take a look at the data for this particular book that has many authors and notice how the `authorsAll` field has the combination of `author` and `authorOthers` (even though our source data didn't have an `authorsAll` field.)

```
$ curl 'http://localhost:8983/solr/bibdata/select?q=id:00000154'

  #
  # notice that authorsAll combines the data from author and authorsOther
  #
  # {
  #   "id":"00000154",
  #   ...
  #   "author":"Kropotkin, Petr Alekseevich,",
  #   ...
  #   "authorsOther":["Brandes, Georg,",
  #     "Agassiz, George R."],
  #   ...
  #   "authorsAll":["Kropotkin, Petr Alekseevich,",
  #      "Brandes, Georg,",
  #      "Agassiz, George R."],
  #   ...
  # }
  #
```

Likewise, let's search for books authored by "George" on the subject of "Throat" using our new `authorsAll` field (`q=authorsAll:george AND subjects:Throat`):

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,author,authorsAll,subjects&q=authorsAll:george+AND+subjects:Throat'

  #
  # response will include
  # {
  #   "id":"00006747",
  #   "author":"Ballenger, William Lincoln,",
  #   "authorsAll":["Ballenger, William Lincoln,",
  #     "Wippern, Adolphus George,"],
  #   "subjects":["Eye", "Ear", "Nose", "Throat"]
  # }
```

```
  #
```

notice that the result includes a book where “George” is one of the authors (even if he is not the main author.)

Another benefit of our customized `author` field is that, because we made it *string* and *single value*, we now can sort the results by author, for example:

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

  #
  # results will be sorted by author
  #
```

Testing changes to the title field

Now run a query for books with the title “run” (`q=title:run`):

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

  #
  # response will include
  #   ...
  #   {"title":"Marco's run /"},
  #   {"title":"50 trail runs in Southern California /"},
  #   {"title":"River running : canoeing, kayaking, rowing, rafting /"}
  #   ...
  #
```

notice that results include books with the title “run”, “runs”, and “running”. This is thanks to the PorterStem filter that the `text_en` field type is using.

Similarly, run a query for books with title `its a dogs new york`:

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=title&q=title:"its+a+dogs+new+york"'

  #
  # response will include
  #   ...
  #   {"title":"It's a dog's New York /"}
  #   ...
  #
```

notice that the results include a book titled “It’s a dog’s New York”. This book was considered a match for our search because the `text_en` field type uses the EnglishPossessive filter that drop the trailing `'s` from the search terms which allowed Solr to find a match despite the poor spelling used in our search terms.

Important: Sorting a `string` field in Solr works as you would expect it, however sorting a `text_en` or `text_general` field field **does not** work because the value has been tokenized. As an exercise try sorting results by `title` (a `text_en` field) and by `title_s` (a `string` field) and compare the results:

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


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

Dynamic Fields

If look at the data in the source `books.json` file you'll notice that some of the records have a property named `urls_ss` that includes a list of URLs for the given book. For example the book with ID 17 has the following data:

```
{
  "id":"00000017",
  "author":"Tabb, John B.",
  "publisher":"Boston,",
  "urls_ss":["http://hdl.loc.gov/loc.gdc/scd0001.00162561418"],
  "subjects":["Children's poetry"]
}
```

You might be wondering what's the significance of the `_ss` in the URLs field. Why didn't we just name that property `urls`?

Solr supports the concept of something called "dynamic fields". Dynamic fields are fields that we define in Solr without giving them a full name, instead we indicate a pattern for the name of the field and Solr will automatically use that pattern for any field that we index that matches the pattern.

For example our `bibdata` core has already a set of dynamic fields defined. You can see them with the following command:

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

  #
  # response will include one that looks like this
  #   {
  #     "name":"*_ss",
  #     "type":"strings",
  #     "indexed":true,
  #     "stored":true},
  #
```

notice that one of the definitions has the pattern `*_ss` which means any field that ends with `_ss` will be assigned the type `strings`, be indexed, and stored. You can dig further and figure out what the definition for the field type `strings` looks like with the following command:

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

  #
  # response will include
  # {
  #   "fieldType": {
  #     "name":"strings",
  #     "class":"solr.StrField",
  #     "sortMissingLast":true,
  #     "docValues":true,
  #     "multiValued":true
  #   }
  # }
```

What this means for us is that, when we imported the data from our JSON file, Solr automatically assigned the type `strings` to the `urls_ss` field because it matched the `*_ss` pattern.

There are lots of pre-defined dynamic fields in a standard Solr installation and you can also define your own dynamic fields.

Take a look at the dynamic fields defined in the `schema.xml` for these projects:

- 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/orangelight/conf/schema.xml

Notice the `*_tesim` vs `*_sim` dynamic field definitions in the Penn State configuration, the `*_sort` and `ignored_*` dynamic field definitions in Princeton's configuration, and the `*_display` vs `*_sort` definitions in Brown's configuration.

subjects_str field (optional)

Of the fields in the schema there are a few of them that look like the values in our JSON file but are *not* identical, for example there is a field named `subjects` and another `subjects_str` but we only have `subjects` in the JSON file. Where does `subjects_str` come from?

When we imported our data, because our `bibdata` core allows for automatic field creation, Solr guessed a type for the `subjects` field based on the data that we imported and assigned it `text_general` but *it also created a separate field* (`subjects_str`) to store a `string` version of the title values. This other field was created as a **copyField**. We can view its definition via the following command:

```
$ curl localhost:8983/solr/bibdata/schema/copyfields?dest.fl=subjects_str

  # response will include
  #
  # {
  #   "source":"subjects",
  #   "dest":"subjects_str",
  #   "maxChars":256
  # },
  #
```

This definition indicates that first 256 characters of the `subjects` will be copied to `subjects_str` field but it does not really tell us what kind of field `subjects_str` will be.

However, the `_str` suffix in the name is a common pattern used in Solr to identify **dynamicFields**. Dynamic Fields are used to tell Solr that any field name imported that matches a particular pattern (e.g. `*_str`) will be created with a particular field definition. Let's look at the default definition for the `*_str` pattern:

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

  # response will include
  #
  # "dynamicField":{
  #   "name":"*_str",
  #   "type":"strings",
  #   "docValues":true,
  #   "indexed":false,
  #   "stored":false
  # }
  #
```

Notice how the `*_str` dynamic field definition will create a `strings` field for any field that ends with `_str`. You can see the definition of the `strings` field type via `curl localhost:8983/solr/bibdata/schema/fieldtypes/strings`

With all this information we can conclude that the first 256 characters of the `subjects` will be copied to a `subjects_str` field (via a **copyField**). The `subjects_str` will be created of the `*_txt` **dynamicField** definition as a `strings` type. `strings` in turn is a multi-value `string` field type.

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'
```

In some instances we passed rather sophisticated values for these parameters, for example we used `q=title:"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.

The selection of the parser has practical implication for us as developers of end user applications. For example, by using the eDisMax parser the syntax of the queries that we pass to Solr is a bit more user friendly than if we use the Standard parser.

For example, let's say that we want to search for all books where "George Washington" exists in the `title` or in the `authorsAll` field, and we want to rank higher books where "George Washington" was the author than when he is referenced in the title.

Using the *Standard* query parser we will need to pass all this information in the `q` parameter to Solr as follows: `q=title:"george washington" authorsAll:"george washington"^10` which we could easily do as developers but it would be rather outrageous to ask an end user to enter such syntax.

By using the *eDisMax* parser we could pass a much simpler `q` parameter to Solr `q="george washington"` and send a separate parameter to configure the fields to search on and their boosting value `qf=title authorsAll^10`. This is possible because the eDisMax parameter supports the `qf` parameter but the Standard parameter does not.

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 noticed, 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` includes the word "washington" (`q=title:washington`)

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

- Retrieve the `id` and `title` (`fl=id,title`) where the title includes the words "women writers" but allowing for a word in between e.g. "women nature writers" (`q=title:"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:"women+writers"~1&fl=id,title'
```

- Documents that have a main author (`q=author:*` means any author)

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

- Documents that do *not* have an author (`q=NOT author:*` means author is not present)

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

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

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

- Documents where title include "story" *and* at least one of the subjects is "women" (`q=title:story AND subjects: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,subjects&q=title:story+AND+subjects:women'
```

- Similar to the previous query, documents where title include "story" (`q=title:story`) *and* at least one of the subjects is "women" (`fq=subjects: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,subjects&q=title:story&fq=subjects:women'
```

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

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title,subjects&q=title:american+AND+-
title:story'
```

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 authorsAll^10`). Remember to select the eDisMax parser (`defType=edismax`) when using the `qf` parameter.

```
$ curl 'http://localhost:8983/solr/bibdata/select?
fl=id,title,authorsAll&q="george+washington"&qf=title+authorsAll^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 authorsAll^-20`). 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:west+AND+authorsAll:nancy&fl=id,title,authorsAll&defType=edismax&debugQuery=on'

  # response will include
  # {
  #   "responseHeader":{...}
  #   "response":{...}
  #   "debug":{
  #     "querystring":"title:west AND authorsAll:nancy",
  #     "parsedquery":"+(+title:west +authorsAll:nancy)",
```

```
  #     "parsedquery_toString":"+(+title:west +authorsAll:nancy)",
  #     "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` for "school" and `_text_` for "teachers") * 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` or the `author` have the word "west", we would use `q=title:west author:west`

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title,author&q=title:west+author: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:west+author:west^5` (notice the `^5` to boost the `author` field)

```
$ curl 'http://localhost:8983/solr/bibdata/select?fl=id,title,author&q=title:west+author: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,author&q=title:west+author: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` field as the default parameter we could pass `qf=title: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` fields it will filter on the *terms indexed* not on the value of the field.

Minimum match (advanced)

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`) 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&mm=3%3C50%25&q=school%20teachers%20secondary%20classroom&qf=title'

  #
  # results will include
  #
  # {
  #   "id":"00010001",
  #   "title":["Succeeding in the secondary classroom : strategies for middle and high school
teachers /"]},
  # {
  #   "id":"00002200",
  #   "title":["Aids to teachers of School chemistry."]},
  # {
  #   "id":"00020157",
  #   "title":["Standards in the classroom : how teachers and students negotiate learning /"]},
  # {
  #   "id":"00008378",
  #   "title":["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_str` as in the following example:

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

  # result will include
  #
  # "facet_counts":{
  #  "facet_queries":{},
  #  "facet_fields":{
  #    "subjects_str":[
  #      "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_str`) to generate the facets rather than the `text_general` version stored in the `subjects` field. This is because, as the Solr Reference Guide indicates facets are calculated "based on indexed terms". The indexed version of the `subject` field is tokenized whereas the indexed version of `subject_str` is the entire string.

You can indicate more than one `facet.field`, for example to get facets for publisher and subjects we would pass `facet.field=subjects_str&facet.field=publisher_str`

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 publishers in the facets rather than all of them you can indicate this with the following syntax: `f.publisher_str.facet.limit=20`. You can also filter only get facets that have *at least* certain number of matches, for example only publishers that have at least 100 books `f.publisher_str.facet.mincount=100` as shown the following example:

```
$ curl 'http://localhost:8983/solr/bibdata/select?
q=*&facet=on&facet.field=publisher_str&f.publisher_str.facet.mincount=100&f.publisher_str.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_str,publisher_str&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`) or the `title` 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+authorsAll&hl=on'

  #
  # response will include
  #
  # "highlighting":{
  #   "00008929":{
  #     "title":["<em>Michael</em> Jackson /"]},
  #   "00022067":{
  #     "authorsAll":["Chinery, <em>Michael</em>."],
  #     "title":["Partners and parents / by <em>Michael</em> Chinery."]},
  #   "00011434":{
  #     "authorsAll":["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` and/or `title`) 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`, `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` field type, 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_general'

  #
  # "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:"twentieth century"`:

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

  #
  # result will include 29 results
  #
```

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

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

  #
  # result will include 33 results
  #
```

More info

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/solr_config.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 author</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, author` (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^10
    authorsAll^5
    subjects
  </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>
    <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` field instead.

Once these changes have been made to the `solr_config.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&q=title: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: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.

Solr Replication

Replication is a technique in which you "create multiple identical copies of your index and load balance traffic across each of the copies" Solr in Action, p. 375.

Solr supports replication out of the box and this is helpful to increase fault tolerance in systems. The basic idea is that if a server becomes unavailable a different server, with an exact copy of the data, can be used to handle search requests. You can also distribute the load between multiple servers at all times.

Solr uses the *one-master, many-replica* model to handle replication in which there is a single *master* server where documents are indexed and multiple *replica* servers where searches can be executed. The replica servers pull data from the master server on a regular basis.

```
Fundamentally, replication requires a server to perform indexing (the master server),
and a server to pull a copy of the index from the master (the [replica] server).
- Solr in Action, p. 376
```

To configure replication in Solr we need to add a new request handler in our `solrconfig.xml` and by convention this handler is named `/replication`.

Master server configuration

To define the server that will act as the *master* in our replication we will add the following handler to `~/solr-7.4.0/server/solr/bibdata/conf/solrconfig.xml` inside the `<config>` element:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="master">
    <str name="replicateAfter">optimize</str>
    <str name="backupAfter">optimize</str>
    <str name="confFiles">managed-schema,stopwords.txt</str>
  </lst>
</requestHandler>
```

Notice the settings under the `master` section. These setting designate this server as the master server in the replication, indicate when the replication will take place, and what configuration files will be replicated. For more details on what each of the settings mean take a look at the Solr Reference Guide

If you reload this core

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

and look at its setting under the Replication tab in the Solr Admin web page you should see a green checkbox next to the "replication enable" setting.

Replica server configuration

For replication to work we need to have other Solr core (or cores) where the data will be replicated to. In a production setting these replica copies will typically live in a different machine from the master, but to keep things simple on this tutorial we are going to define the replica on the same machine and Solr installation that we have been using.

Let's start by defining a new core:

```
$ solr create -c theothercore
```

If you query `theothercore` it will have no documents since it is brand new.

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

  #
  # {
  #  "responseHeader":{
  #    "status":0,
  #    "QTime":3,
  #    "params":{
  #      "q":"*"}},
  #  "response":{"numFound":0,"start":0,"docs":[]
  #  }}
  #
```

Let's configure `theothercore` to be the replica our `bibdata` core. We do this by adding a `/replication` handler to the `solrconfig.xml` of our new core, in other words to the file at `~/solr-7.4.0/server/solr/theothercore/conf/solrconfig.xml`. The replication handler in this case will be marked as "slave", rather than "master", as indicated below. Add the following inside the `<config>` element:

```
<requestHandler name="/replication" class="solr.ReplicationHandler">
  <lst name="slave">
    <str name="masterUrl">http://localhost:8983/solr/bibdata/replication</str>
    <str name="pollInterval">00:00:20</str>
    <str name="compression">internal</str>
    <str name="httpConnTimeout">5000</str>
    <str name="httpReadTimeout">10000</str>
    <str name="httpBasicAuthUser">username</str>
    <str name="httpBasicAuthPassword">password</str>
  </lst>
</requestHandler>
```

Notice that in these settings we indicate the URL of the master Solr index, in our case `http://localhost:8983/solr/bibdata/replication`. This is how the replica knows where to get the data from. We also indicated that we want the replicate to poll the master server every 20 seconds (`00:00:20`) for changes in the data (we will use a different interval if we were in production.)

Let's reload reload this core for the changes to take effect:

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

If we query the replica again we would see documents on it. Also, if we look at the settings under the Replication tab in the Solr Admin web page *for this new core* we would see information about the replication including the master URL and the polling interval.

Sharding

Sharding is a different technique from replication that is used to execute *distributed queries* across multiple Solr cores. This is

useful when you have too many documents to handle on a single server.

Sharding is beyond the scope of this tutorial but Chapter 12 of Solr in Action or the Solr Reference Guide are good places to start if you want to learn more about it.

Trivia: “One of the upper limits in Solr is that an index cannot contain more than 2^31 documents, due to an underlying limitation in Lucene.” (Solr in Action, p. 372). That’s two billion documents (2,000,000,000) for those of us that use the short scale.

SolrCloud

Solr also provides a set of features known as SolrCloud that is the preferred way to handle fault tolerance and high availability in large scale environments. The Solr Reference Guide defines it as:

```
SolrCloud is flexible distributed search and indexing, without a master node
to allocate nodes, shards and replicas. Instead, Solr uses ZooKeeper to manage
these locations, depending on configuration files and schemas. Queries and
updates can be sent to any server. Solr will use the information in the
ZooKeeper database to figure out which servers need to handle the request.
```

This is also out of the scope of this tutorial.

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 the 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.