SOLR Steps to build the foundations for PaperSearch

Step 1: Start the Solr server using the solr script stored under bin (in other words, **bin/solr**). The following commands are used to start, stop and find the status of the server. Solr, by default, runs on port 8983. But it can be started on another port.

```
ashwath@kilda:~/solr-7.3.0$ bin/solr start

ashwath@kilda:~/solr-7.3.0$ bin/solr status

ashwath@kilda:~/solr-7.3.0$ bin/solr stop
```

Step 2: Create managed_schema, solrconfig.xml, papers_data_config.xml inside a configset called 'papers' under server/solr/. Details of how to create these files are given in the appendix.

Step 3**: Create a core/collection** (2 names for the same thing) by running the **bin/solr** script. The **-c flag** indicates that a collection is being created. The **-d flag** is used to specify a **configset** (created in Step 2). Note that this configset is used to create the collection at server/solr, where the data is also stored.

If the **-d flag** is not set, solr creates a default schema (there is a configset called **_default** which is automatically used). Solr calls this 'schemaless mode'. The process is shown for three different indices (third one is schemaless).

It's pretty easy to **delete** a collection too. This is shown in the 4th command.

```
ashwath@kilda:~/solr-7.3.0$ bin/solr create -c papers -d
server/solr/configsets/papers/

ashwath@kilda:~/solr-7.3.0$ bin/solr create -c references -d
server/solr/configsets/references/

ashwath@kilda:~/solr-7.3.0$ bin/solr create -c metadata

ashwath@kilda:~/solr-7.3.0 bin/solr delete -c papers
```

Step 4: Now that the collection has been created, it's time to insert the data. There are 3 options for doing this, all of which I'll describe.

1. Use the bin/post tool
2. Use the data import handler
3. Use the python library pysolr.

I started with the bin/post tool. It has some disadvantages, but it's pretty easy to use, and doesn't need much configuration (unlike the Data Import handler).

INSERTING DATA USING THE BIN/POST TOOL:

In general, the **bin/post** command for delimiter-separated files is like this

```
ashwath@kilda:~/solr-7.3.0$ bin/post -c papers /tmp/es/ -type text/csv -params
"fieldnames=sentence&separator=%C2%B8"
```

There are plenty of limitations which I will not describe in this document as I have described them elsewhere. But in general, bin/post can be replaced by one of the other methods.

Let's turn our attention to the second method: using the Data Import Handler.

Data Import Handler:

Note: this assumes that the collection has already been created using bin/solr

I wanted to get the filename into the solr index. The Data Import Handler can be used for this, while bin/post does not have this capability. Of course, this might not be important in most cases.

The first thing to do to use the data import handler is to define the data config file (given in appendix). Once that is done, the following command has to be run (the appropriate collection has to be specified here).

```
curl 'http://localhost:8983/solr/references/dataimport?command=full-import&clean=true'
```

View schema:

```
curl http://localhost:8983/solr/references/schema
```

Indexing JSON files using bin/post

The Data Import handler cannot be used with json files. One option is to fall back to bin/post, which is what I did at the beginning. NOTE: I ended up reindexing using Pysolr, so I'm not using this anymore. It is impossible to get the filename into the index using this method.

Step 1: Create the core after creating the config files. No need of data_config, as Data Import Handler doesn't work with json files.

Note: If you need to update fields through the Schema API, use something like this:

```
curl -X POST -H 'Content-type:application/json' --data-binary '{"add-field":{"name":url", "type":"string", "multiValued":false, "stored":true}}' http://localhost:8983/solr/films/schema
```

This is usually used when you use the default schema (schemaless mode) and want to change datatypes and things like that. But I'm creating the schema manually instead, I just find that much cleaner.

Step 2: Call bin post on each json file. The function is needed because bash complains about the length of the expansion while using *.json.

```
function run_post ()
{       while read line1; do
                bin/post -c metadata /tmp/metasolr/$line1
        done
}
$ run_post
```

Pysolr

Getting the filename in the Solr index isn't possible with bin/post, and the Data Import handler is out of bounds for Json.

Therefore, I turned to Pysolr. Even though it has awful documentation, it is probably the easiest method when using Python. Of course, the other 2 methods are programming-language independent.

Indexing Json metadata using Pysolr:

The first thing to do is to define the fields and fieldnames in managed_schema. The default solrconfig is all that is required, there is no need of any changes there. Once this is done, the files can be indexed easily using the following Python code.

```
def insert_metadata_into_solr():
    solr = pysolr.Solr('http://localhost:8983/solr/metadata')

    basepath = '/home/ashwath'
    folderpath = os.path.join(basepath, 'arxiv-cs-dataset-LREC2018')
    list_for_solr = []
    for filepath in iglob(os.path.join(folderpath, '*.meta')):
        with open(filepath, 'r') as file:
            filename = os.path.basename(filepath)
            filename_without_extension = '.'.join(filename.split('.')[:2])
            content = json.load(file)
            #print(content['title'], content['authors'], content['url'], filename)
        solr_content = {}
        solr_content['authors'] = content['authors']
        solr_content['title'] = content['title']
        solr_content['url'] = content['url']
        solr_content['filename'] = filename_without_extension
        list_for_solr.append(solr_content)
        #solr.add([solr_content])

    solr.add(list_for_solr)
```

There's just a solr add method which is used to add a list of dictionaries. Each dictionary corresponds to one record. The fields defined in managed_schema are supplied as keys in this dictionary, and the values are the values we want to see in the Solr index.

Note that the iglob submodule of glob is very useful used in the above code allows UNIX-list wildcards while reading files. The os-path.basename can be used to easily identify the filename of the current file being read, and this can be passed to the index in Solr.

And finally, building the list of dictionaries and THEN sending it to SOLR is way faster than sending each individual dictionary to Solr as it is created.

APPENDIX:

CONFIGURATION:

There are 2 options (well, 3 actually if you include working on the web UI) for creating the solr config file. Every time you index something, you need 2 files mandatorily: solrconfig.xml and managed_schema.

1. You can use 'schemaless mode' and let solr generate a default config (and default managed_schema).
2. But the preferred method (more control to the user) is to create a configset under: server/solr/configsets/<name>.
   Then under this folder, insert a folder conf, and under that we will have our config files: managed_schema, solconfig.xml, and other optional files such as synonyms.txt, stopwords.txt, the lang folder, and very importantly, the data config file.

   I'll split this section into two now, as the configs are slightly different.

PAPERS_TXT_GROUPED:

Data_config file: contains fields to import using the DataImport Handler. There is a 'raw Line' , which is split into five fields.

```
<dataConfig>
 <dataSource name="fds"  type="FileDataSource"/>
  <document>
    <!-- this outer processor generates a list of files satisfying the conditions
         specified in the attributes -->
    <entity name="getfiles" processor="FileListEntityProcessor"
            fileName="arxiv-cs-dataset-LREC2018-grouped.tsv"
            dataSource="null"
            rootEntity="false"
            baseDir="/home/ashwath/"
    >
   <field column="rawLine" name="line"/>
   <entity name="procline"
                processor="LineEntityProcessor"
                url="${getfiles.fileAbsolutePath}"
        dataSource="fds"
        transformer="RegexTransformer" >

   <field column="rawLine"
                 regex="^(.*)\t(.*)\t(.*)\t(.*)\t(.*)$"
                 groupNames="filename,date,title,sentencenum,sentence"
          />
    </entity>
   </entity>
  </document>
</dataConfig>
```

UPDATE solrconfig.xml

Solr config has everything needed by default, just one change is required to use the Data Import request Handler: Add this.

```
    <!-- Data Import Handler -->


    <requestHandler name="/dataimport"
class="org.apache.solr.handler.dataimport.DataImportHandler">
      <lst name="defaults">
        <str name="config">papers-data-config.xml</str>
      </lst>
    </requestHandler>
```

Managed schema is where all the magic happens.

1. **Add the fields**. These fields all have a datatype. There are default datatypes, but we can also define our own. Text_classic is a datatype I defined with specific tokenizers and filters. We can choose to index and store or not. Line is not going to be used in the query or the results, so I'm not storing or indexing it.

```
<field name="line" type="text_classic" indexed="false" stored="false"/>
<field name="fileName" type="string" indexed="true" stored="true"/>
<field name="date" type="pdate" indexed="true" stored="true"/>
<field name="title" type="text_classic" indexed="true" stored="true"/>
<field name="sentencenum" type="pint" indexed="true" stored="true"/>
<field name="sentence" type="text_classic" indexed="true" stored="true"/>
```

2. **Add the field types:** I have just shown text_classic and a couple of useful defaults.

```
<!-- A text field that only splits on whitespace for exact matching of words -->
<dynamicField name="*_ws" type="text_ws"  indexed="true"  stored="true"/>
<fieldType name="text_ws" class="solr.TextField" positionIncrementGap="100">
  <analyzer>
    <tokenizer class="solr.WhitespaceTokenizerFactory"/>
  </analyzer>
</fieldType>

<!-- A general text field that has reasonable, generic
     cross-language defaults: it tokenizes with StandardTokenizer,
       removes stop words from case-insensitive "stopwords.txt"
       (empty by default), and down cases.  At query time only, it
       also applies synonyms.
  -->
<fieldType name="text_general" class="solr.TextField" positionIncrementGap="100"
multiValued="true">
  <analyzer type="index">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymGraphFilterFactory" synonyms="index_synonyms.txt"
ignoreCase="true" expand="false"/>
    <filter class="solr.FlattenGraphFilterFactory"/>
```

```
    -->
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.StandardTokenizerFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
/>
    <filter class="solr.SynonymGraphFilterFactory" synonyms="synonyms.txt"
ignoreCase="true" expand="true"/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Here, in text_classic, I tokenize using the Classic factory, so that words with hyphens are retained and not tokenized, urls are not tokenized. Both these were requirements, as the input file has text like GC:xxx, and http://dblp/... Data, which aren't to be tokenized. The StopFilterFactory uses English stopwords (in the last line, with ignore case =true) and removes them. Lowercase filter makes everything lowercase, and the Porter Stemmer does stemming.

I have added two separate analyzers for the query and the index. This is optional, you can also have just one common analyser for querying and for indexing:

```
    <fieldType name="text_classic" class="solr.TextField"
positionIncrementGap="100" multiValued="true">
      <analyzer type="index">
        <tokenizer class="solr.ClassicTokenizerFactory"/>
        <filter class="solr.StopFilterFactory" ignoreCase="true"
words="stopwords.txt" />
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.PorterStemFilterFactory"/>
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"
        />
      </analyzer>
      <analyzer type="query">
        <tokenizer class="solr.ClassicTokenizerFactory"/>
        <filter class="solr.LowerCaseFilterFactory"/>
        <filter class="solr.PorterStemFilterFactory"/>
        <filter class="solr.StopFilterFactory"
                ignoreCase="true"
                words="lang/stopwords_en.txt"
        />
      </analyzer>
    </fieldType>
```

References:

It has the following data config. Notice that we get the filename directly and store it in a field. This field must match a field declared in managed schema

```
<dataConfig>
 <dataSource name="fds"  type="FileDataSource"/>
  <document>
    <!-- this outer processor generates a list of files satisfying the conditions
```

```
        specified in the attributes -->
    <entity name="getfiles" processor="FileListEntityProcessor"
            fileName="^.*\.csv$"
            dataSource="null"
            rootEntity="false"
            baseDir="/tmp/solrrefs"
    >
   <field column="file" name="fileName"/>

   <entity name="procline"
                processor="LineEntityProcessor"
                url="${getfiles.fileAbsolutePath}"
        dataSource="fds"
        transformer="RegexTransformer" >

   <field column="rawLine"
                 regex="^(.*);(.*);$"
                 groupNames="annotation,details"
         />

    </entity>
   </entity>
  </document>
</dataConfig>
```

Update **solrconfig** similarly to the previous case.

Update managed_schema.

```
<field name="annotation" type="string" indexed="true" stored="true"/>
<field name="authors" type="text_classic" indexed="true" stored="true"/>
<field name="details" type="text_classic" indexed="true" stored="true"/>
<field name="dummy" type="text_classic" indexed="true" stored="true"/>
<!--<field name="filename" type="string" indexed="true" stored="true"/> -->
<field name="fileName" type="string" indexed="true" stored="true" />
<field name="rawLine"  type="text_classic" indexed="false" stored="false"/>
<fieldType name="text_classic" class="solr.TextField" positionIncrementGap="100"
multiValued="true">
  <analyzer type="index">
    <tokenizer class="solr.ClassicTokenizerFactory"/>
<filter class="solr.ClassicFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
/>
    <!-- in this example, we will only use synonyms at query time
    <filter class="solr.SynonymGraphFilterFactory" synonyms="index_synonyms.txt"
ignoreCase="true" expand="false"/>
    <filter class="solr.FlattenGraphFilterFactory"/>
    -->
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.ClassicTokenizerFactory"/>
<filter class="solr.ClassicFilterFactory"/>
    <filter class="solr.StopFilterFactory" ignoreCase="true" words="stopwords.txt"
/>
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

Metadata:

I first used the schemaless mode for this index, but it's always better to do it manually. These are JSON files, so the Data Import Handler doesn't work.

I indexed first using bin/post, but then I found out pysolr really makes things much easier.

PYSOLR:

To index the metadata files with 3 fields url, authors, title (and the arxiv_identifier from the filename), follow the steps below.

Add these fields in **managed_schema**. And the same text_classic fieldtype.

```
<field name="url" type="string" indexed="true" stored="true" multiValued="false"/>
<field name="authors" type="text_classic" indexed="true" stored="true"
multiValued="true"/>
<field name="title" type="text_classic" indexed="true" stored="true"
multiValued="false"/>
<field name="arxiv_identifier" type="string" indexed="true" stored="true"
multiValued="false"/>
```

No changes required to solrconfig.xml.

Now in Python, we need to read the files one by one (using the excellent iglob module of the glob package, which allows unix-like file reading – wildcards and the like), get the fields and insert them into a dictionary.

While sending the data to Solr, pysolr expects a list of dictionaries where each dictionary is a record.

But before that, we need to get the connection to Solr.

```
solr = pysolr.Solr('http://localhost:8983/solr/metadata')
```

Here's my complete code from the previous section.

```
import json
import os
import pysolr
from glob import iglob

def insert_metadata_into_solr():
    solr = pysolr.Solr('http://localhost:8983/solr/metadata')

    basepath = '/home/ashwath'
    folderpath = os.path.join(basepath, 'arxiv-cs-dataset-LREC2018')
    list_for_solr = []
    for filepath in iglob(os.path.join(folderpath, '*.meta')):
        with open(filepath, 'r') as file:
            filename = os.path.basename(filepath)
            filename_without_extension = '.'.join(filename.split('.')[:2])
            content = json.load(file)
            #print(content['title'], content['authors'], content['url'], filename)
        solr_content = {}
        solr_content['authors'] = content['authors']
        solr_content['title'] = content['title']
```

```
        solr_content['url'] = content['url']
        solr_content['filename'] = filename_without_extension
        list_for_solr.append(solr_content)
        #solr.add([solr_content])

    solr.add(list_for_solr)

if __name__ == '__main__':
    insert_metadata_into_solr()
```

I repeat. In my code, I could have inserted each dictionary into solr straightaway. But this is very inefficient and is slow for even moderate amounts of data. It is much better to build a list of dictionaries and then insert everything into Solr in one go.

Arxiv_metadata

The arxiv_metadata index uses a similar set of fields in managed_schema. However, the Python code is more complex as this involves fetching the published date from the XML file from Arxiv. So the program to do this has to parse this XML file (I use the lxml module).

Here is the complete code to parse the XML, get the dates for all the records, and then insert it into the Solr index. It has a lot of inline comments which explain the steps in the code.

```
from collections import defaultdict
from lxml import etree
import pysolr
# Parse the Arxiv xml file
def get_xml_root():
    """ Gets the root of the arxiv xml tree and returns it."""
    xml_filepath = '/home/ashwath/Files/arxiv-cs-all-until201712031.xml'
#     xml_filepath = 'D:\\Coursework\\HiWi\\arxiv-cs-all-until201712031.xml'
    doc = etree.parse(xml_filepath)
    root = doc.getroot()
    return root

def parse_xml_insert_into_solr(root):
    """ Function which parses the arxiv xml, and inserts some of the
metadata
    into an index in Apache Solr."""
    # Set the 2 namespaces which are used in the xml file: Open archive,
    # and Dublin Core.
    namespace = {'dc': 'http://purl.org/dc/elements/1.1/',
                 'oai_dc': 'http://www.openarchives.org/OAI/2.0/oai_dc/'}
    # NOTE: this is the fully qualified version of descending through the
ns
    # for m in root.find('./record/metadata/'
    # '{http://www.openarchives.org/OAI/2.0/oai_dc/}dc/'
    #'[{http://purl.org/dc/elements/1.1/}identifier=
    # "http://arxiv.org/abs/0704.0002"]'):

    # Descend to the metadata node in all the records: it has all the
    # interesting fields as its children.
    metadata_xpath = './record/metadata/'
    # metadata is a list, but len(metadata) = len(root) = 155308.
    # Each member of metadata, metadata[i] is of type lxml.etree._Element
node,
```

```
    # the same as root. So we can loop through it to get its children.
    metadata = root.findall(metadata_xpath, namespaces=namespace)
    #print(len(metadata))
    # Get the index of the first character after the dc prefix, it's going
to
    # be the same for all children. {http://purl.org/dc/elements/1.1/}title
    # Get the tag of the first record's title, and run find on that. Ans.
34
    tag_with_prefix = metadata[0][0].tag # for first child
    start_index = tag_with_prefix.find('}') + 1 # 34
    list_for_solr = []
    for metadata_element in metadata:
        solr_record = defaultdict(list)
        for child in metadata_element:
            if child.tag[start_index:] == 'title':
                solr_record['title'] = child.text
            elif child.tag[start_index:] == 'creator':
                solr_record['authors'].append(child.text)
            elif child.tag[start_index:] == 'date':
                solr_record['published_date'].append(child.text)
            elif child.tag[start_index:] == 'identifier' and \
              child.text.startswith('http://arxiv'):
                solr_record['url'] = child.text
                id_startindex = child.text.rfind('/') + 1
                solr_record['arxiv_identifier'] =
child.text[id_startindex:]    # Return only the first date (1st element in
list): the date of version 1.
        # Append each record in dict form (not defaultdict) to the
        # list_for_solr list.
        list_for_solr.append(dict(solr_record))
    solr = pysolr.Solr('http://localhost:8983/solr/arxiv_metadata')
    solr.add(list_for_solr)

if __name__ == '__main__':
    xml_root = get_xml_root()
    parse_xml_insert_into_solr(xml_root)
```