Dumb-down at Indexing
Nested Data in the Solr Search Engine

Sigfrid Lundberg

Digital Transformation
Royal Danish Library
Post box 2149
1016 Copenhagen K
Denmark

ABSTRACT

This is a discussion of how to index and present metadata with possibly innumerable number of fields. We reject the idea to use the joins and hierarchically structured documents in a search engine like SOLR.

Instead we turn to the dumb down concept used by metadata communities to describe what goes on when crosswalking a complex metadata system with a more shallow, less detailed one.

Introduction

The Royal Danish Library has been using the Solr (https://solr.apache.org/) search engine for at least a decade. Almost all projects that need some search facilities are using it. A Swiss army knife for searching in small as well as well as big data sets. A trusty tool that provide many advantages to the alternative to use a relational database management system (RDBMS) when working with resource discovery systems.

At the first Solr workshop I participated the teacher reiterated over and over again that Solr is a search engine, not a data management tool. Although Solr can Create, Retrieve, Update and Delete (CRUD) documents, the transactions cannot really be characterized as having Atomicity, Consistency, Isolation and Durability (ACID)

The role of bibliographic structure

In a library we encounter data with much more structure than simple packages of attributes and values, but also much less structure than what you expect from data in [database normal form](https://en.wikipedia.org/wiki/Database_normalization). Bibliographical data may **for instance** describe texts that

- have one or more titles, each having different type (main title, subtitle, translated title, transcribed title, uniform title to iterate some of the frequently used ones)
- may have one or more authors that may be persons or organizations
- each of which may have dates of birth and death
- and an affiliation

If we were using a RDBMS, the data on persons could be stored in one database table, the titles in another and there could be a third table keeping track (through joins and foreign keys) of each person's relations to the works to which they've contributed. Someone made the illustrations, someone else wrote the texts. A third one made the graphical design.

For a portrait photograph we have one person being the photographer, and another being the subject. The data on the subject can be as important as the data on the artist.

These data are important. For instance the dates are used for distinguishing between people with the same name. (The digital phone book krak.dk lists 43 now living persons named *Søren Kierkegaard* , while *the philosopher* died 1855). An importand use case is how to decide whether a given object is free from copyright or not, such as when the originator has been dead for more than 75 years.

Add to this that there are many more complications, like we may have multiple copies of a given title and that each copy may belong to different collections with very different provenance. For historical objects we may even have a manuscript in the manuscript and rare books collections and a modern pocket book in the open stocks.

The role of the primary purpose of these data are to enable library patrons and service users to search and retrieve. Typically users type a single word in the search form, while ignoring the handful of fields that we provide through the effort of catalogers, software developers etc.

The user gets a far too long list of results which contains author, title and perhaps some subjects or keywords. Again typically, the user clicks on a title an gets a landing page. This is far more detailed; it presents many fields that will enable users to decide whether to order or retrieve the object.

If we grossly simplify the process, a user might be up to one of two things: (1) Finding or resolving a given reference, i.e., the right object, perhaps the right edition. Or (2) Finding information on a given topic. See these two as endpoints on a scale.

You may look for a particular book *Enten — Eller (Either/Or)* by *Kierkegaard* (1843) or you might be interested in the role of this philosopher in your study of the origin of existentialism. In the former case you actually look for Kierkegaard in the author field, in the latter case you look for him in subject field.

Encoding, indexing and using

Assume we are about to add metadata on *Enten — eller* by *Søren Kierkegaard* into a Solr index. What we get for that book might contain the data below. The example is encoded in a format called Metadata Object Description Schema (MODS). Note that the namespace prefix ‘md‘ stands for the URI ‘http://www.loc.gov/mods/v3‘

Enten — Eller.

Et Livs-Fragment

udgivet

af

Victor Eremita.

Første Deel

indeholdende A.'s Papirer.

Er da Fornuften alene bedt,
ere Lidenskaberne Hedninger?

Young.

Kjøbenhavn 1843.

Faaes hos Universitetsboghandler C. A. Reitzel.

Trykt i Bianco Lunos Bogtrykkeri.

```
<md:titleInfo>
   <md:title>Enten - eller</md:title>
   <md:subTitle>Et livsfragment</md:subTitle>
</md:titleInfo>
<md:name displayLabel="Author"
         type="personal"
         authorityURI="https://viaf.org/viaf/7392250/"
         xml:lang="en">
   <md:namePart type="family">Kierkegaard</md:namePart>
   <md:namePart type="given">Søren</md:namePart>
   <md:namePart type="date">1813/1855</md:namePart>
   <md:alternativeName altType="pseudonym">
      <md:namePart>Victor Eremita</md:namePart>
      <md:description>“Victorious hermit,” general editor of
      Either/Or, who also appears in the first part of its
      sequel, Stages on Life’s Way. Also the author of the
      satirical article “A Word of Thanks to Professor
      Heiberg.”</md:description>
   </md:alternativeName>
   <md:role>
      <md:roleTerm type="code">aut</md:roleTerm>
   </md:role>
</md:name>

<md:originInfo>
   <md:dateCreated>1843</md:dateCreated>
   <md:publisher>Universitetsboghandler C. A. Reitzel</md:publisher>
</md:originInfo>
```

This is a fake record I created for the purpose of this paper. The work has a *title* and a *name* , which has a *role* (which is *aut* as in author). The *name* also has multiple parts like family name (Kierkegaard), given name (Søren) and a date (1813/1855 which is ISO's way to express a date range i.e., the years between which the philosopher was alive. His *Fragment of Life*

To get the birth and death dates you have to parse a string. As a matter of fact, the name on the book cover wasn't *Søren Kierkegaard* but *Victor Eremita* (Victorious hermit), encoded as an <alternativeName>. A telling pseudonym of the author of The Seducer's Diary. Søren was good at pseudonyms .

The <md:role> ... </md:role> permits the cataloger to encode that a person called *Søren* has the `aut` relation to the work. Library of Congress have lists with houndreds of such relators . Actually, each of those could be seen as a field. The thing is that even in a large bibliographic database there would be very few records where *Data manager* (`dtm`), *Former owner* (`fmo`) and *Librettist* (`lbt`) would contain any data.

Now we've identified a lot of possible fields to use, for cataloging and for information retrieval. They have perfectly reasonable use cases, and all of them are used in everyday library practice, so how do I get them into my Solr index?

The attempt

We have tried to put such records into Solr. The attempt was successful. In the rest of this paper I will outline how we did that, learn you a bit on how to use such an index and finally why decided not to implement it.

In our experiments we transformed MODS records to nested Solr records, such as the record below, which is transformed from my fake record above.

```
[
  {
   "id": "https://example.org/record",
   "described": true,
   "entity_type": "the_object",
   "cataloging_language": "en",
   "record_created": "2022-08-12",
   "tit": [
        {
          "describing": "https://example.org/record",
          "described": false,
          "entity_type": "title main",
          "title": [
            "Enten - eller"
          ],
        "id": "https://example.org/record!disposable!subrecord!d1e21"
        }
   ],
   "aut": [
      {
          "id": "https://example.org/record!disposable!subrecord!d1e30",
          "authority": "https://viaf.org/viaf/7392250/",
          "described": false,
          "describing": "https://example.org/record",
          "language": "en",
          "entity_type": "aut",
          "agent_name": "Kierkegaard Søren (1813/1855)"
      }
   ],
   "visible_date": ["1843"],
  }
 ]
```


If you are familiar with the workings of Solr, you know that the data-model (if I may label it as such) used is configured in a file call 'schema.xml'. It basically contains list of fields that can be used in what is referred to as 'Solr documents'. In such a schema you may add

```
<field     name="_nest_path_"
           type="_nest_path_"
           stored="true"
           indexed="true" />
<field     name="_nest_parent_"
           type="string"
           indexed="true"
           stored="true" />
```

the former of which is of the following type:

```
<fieldType name="_nest_path_" class="solr.NestPathField" />
```

See the Solr Indexing Nested Child Documents documentation.

The nested indexing works since the indexer stores an xpath like entity for each record, making it possible track which Solr document which is parent and which document which is child which is the parent. That info is in the `_nest_path_` field and Solr does that automatically whenever it starts a new document inside a parent one.

You will get that information back from the server if you add a Solr field list argument (`fl`) at search time

```
fl=*,[child]
```

That is straight forward. The problem is then to make Solr search in the child documents and return the parent or root document.

```
{!parent which="described:true"}
         {!edismax v="agent_name:(Kierkegaard Søren) AND entity_type:aut"}
AND
{!parent which="described:true"}
         {!edismax v="title:(Enten - eller) AND entity_type:tit"}
```

The constructs {!parent ... } and {!edismax ... } are so called local parameters in a Solr request. The former specifies that we want Solr to return parent documents such the described:true, the latter tells Solr we want the author to be Søren and title to be Enten — eller. Now we can reasonably easy search and retrieve information on the *Etcher* (`etr`) and *Dancer* (`dnc`), when applicable.

This is a special case of `join` as implemented in Solr. Recall that joins are at very very core of `SQL` , and one of the features making the RDBMS such a powerful tool.

Also recall that I mentioned that my first Solr instructor dissuaded us from using search engines as data stores. Does that generalize to other features coming from the database world?

The user problems

I hope I've been able to convince you that the fairly complicated metadata structures used in libraries are useful for patrons and staff. They were not invented for giving software developers gray hair and age prematurely. Also, it is legitimate use case to be able to identify the etchers and the dancers.

However:

- We do, however, know that users at of our resources are not very good at using fields. An interface allowing you to search portraiture subjects is very specialized use case. So is the use case to be able to search for senders and recipients of letters.
- People do search for word in a title, but they do not search forA life fragmentseparate fromEither/or. Likewise they not particularly interested in making a difference betweenEnten — ellerandEither/or. If they search for the latter they presumably want an English translation, but when studying a detailed presentation they are almost certainly interested to know that Either/or is actually a translation.
- You know, each performance ofВесна священная(AKA The Rite of Spring) has a conductor, director and choreographer and a lot of dancers, obviously in addition to *Стравинский, Игорь Фёдорович* (AKA *Stravinsky, Igor Fyodorovich* the composer). I could go on here. You could add from your own experience.

To make a useful service we have to aggregate data into reasonable headlines. _[Dublin Core Metadata Initiative](https://www.dublincore.org/)_ has actually a name for this: The [Dumb-Down Principle](https://www.dublincore.org/resources/glossary/dumb-down_principle/)

The developer problems

From the developers point of view, metadata dumb-down can take place, either (i) when indexing or (ii) when searching.

In either case, for a ballet performance we would dumb-down _Composer_ ((`cmp`), Conductor ((`cnd`), Director ((`drt`) and Choreographer ((`chr`) to one single repeatable field [creator](https://www.dublincore.org/specifications/dublin-core/dcmi-terms/terms/creator/). It would contain Igor Stravinsky (the transcribed, but perhaps also his name in Cyrillic), and obviously all other creatives. Most of the dancers would most likely go to the contributor field.

Doing dumb-down at indexing would mean to create fields `creator` and `contributor` in the index, to do it when searching would imply to do it using the horrendous search syntax presented above. Then you have to do the same for title and other relevant fields.

In the case of *Either/or* , *Enten — eller* the dumb-down solr record would look somewhat as the record below:

```
[
   {
   "id": "https:!!example.org!record",
   "title": [
      "Enten  &#8212; eller"
   ],
   "creator": [
      "Kierkegaard, Søren 1813/1855"
   ],
   "record_created": "2022-08-12",
   "visible_date": [
      "1843"
   ],
   "original_object_identifier": [],
   "pages": []
   }
]
```

Hence when indexing we only create one record, and no joins are needed. A query could be

```
creator:kierkegaard AND title:(enten eller)
```

The drawback being that the in the index we cannot tell the difference between *Igor Stravinsky* (`cmp`) and the *Conductor* (`cnd`). Both are creators. The dumbed-down index has lost most of the information you need to decide whether you want to listen to an album or see a performance.

- At indexing: Your search syntax is nice and clean. You have to use some other method to present the data in the detailed view.
- At search: Your search syntax is very complicated. On the other hand, you have all the data needed for the detailed view.
- At a practical level, the nested Solr seems more or less experimental, and the documentation is less than excellent. Only the lucene query parser supports it, and when searching with (for example) `edismax` query parser you run into the syntactic problem with local parameters demonstrated above.

If we are to describe the situation in Model-View-Controller (MVC) terms, the second (i.e., the at search implementation) looks nice. One model, one controller but (perhaps) two views. When doing it at indexing, we need two models and an architecture diagram might look much more messy. Semantic exercise to make the dumb-down scheme might seem complicated. The code, however, is much simplified.

The fact that each substructure in the nested Solr document must follow the same schema is an annoying feature. It isn't important, but persons, subjects and whatever all have the same content model (in the sense of an XML DTD or Schema), makes the setup much less attractive.

Finally, it is my experience that it easier to accommodate multiple metadata models and standards in the same index with dumb-down at indexing. In our case we opted for transforming our MODS records to the schema.org ontology for the detailed presentation. Hence, retrieval will be from a separate datastore. The schema.org ontology is rich enough for our landing pages and detailed result sets. It provides an extra bonus, we hope, in that Google would actually be able to index our collection.

The only advantage I can see with at search time dumb-down is that we would have only a single model in our search application.

Conclusion

In the end, after some weeks work, we threw out our nested indexing stuff and most likely we a threw out some baby we were not aware of with the bathwater. Be that as it may, we opted for an easy format for search, while retaining detail for presentation, and interoperability for other uses.

Library patrons have more needs than resource discovery. Some use APIs for study, research or for services of their own. The search index, schema.org, the original mods will eventually be available for such purposes. It could be that a nested index could actually be useful for such users.

References

ACID. In:*W ikipedia*
https://en.wikipedia.org/wiki/ACID

ConclusivePostscript, 2013. A "Who's Who" of Kierkegaard's Formidable Army of Pseudonyms.
https://www.reddit.com/r/philosophy/comments/1n2opm/a_whos_who_of_kierkegaards_formidable_army_of/

Contributor
https://www.dublincore.org/specifications/dublin-core/dcmi-terms/terms/contributor/

Create, read, update and delete. In:*W ikipedia*
https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

Date
https://www.dublincore.org/specifications/dublin-core/dcmi-terms/terms/date/

Indexing Nested Child Documents. In: *Apache Solr Reference Guide*
https://solr.apache.org/guide/8_1/indexing-nested-documents.html

Library of Congress*Metadata Object Description Schema*
https://www.loc.gov/standards/mods/

Library of Congress*MARC Code List for Relators*
https://www.loc.gov/marc/relators/relacode.html

The Standard Query Parser. In:*Apac he Solr Reference Guide*
https://solr.apache.org/guide/6_6/the-standard-query-parser.html

schema.org
https://schema.org