# Knowledge Discovery of MARCXML based Library Databases using FOSS Full Text Indexing Tools

**Yatrik Patel** **Dinesh Rayka** **Divyakant Vaghela**

## Abstract

*This paper discusses introduction to full text search, its comparison with traditional approach, which gives insight on popular open source indexing and searching tools viz a viz. Lucene and Solr by Apache Foundation. In internet era libraries are known as knowledge warehouses. Library databases are prominently available in MARCXML format and there are very few free and open source (FOSS) implementations are available in public domain. Many efforts are made in integrating the popular searching and indexing techniques which powers even some of the world's largest search engines, In library domain this has not been attempted properly. This paper is an attempt to showcase technique for implementation of Lucene/Solr based full text indexing over MARCXML based catalogue records to provide reliable, fast and dynamic search interface with provision of facets.*

**Keywords:** Fulltext Indexing, Lucene, Solr, MARCXML, Knowledge Discovery, Faceted Search

## 1. Introduction

### 1.1 Definitions: Full Text Search and Indexing

Full text search is basically a technique to search machine readable document or database, In a full text search, the search engine examines all the words in every stored document as to match with search terms supplied by the user. Full text searching techniques are quite common in online bibliographic databases

When one is dealing with small number of document, it is always possible to scan through each document for the search term, but with document store is potentially large, time being taken to perform this search becomes larger and larger, hence decreasing performance, To avoid this phenomena smarter search engines divides this task in to two stages i.e. 'indexing' and 'searching': The indexing stage scans all entire document store and builds list of all possible search terms, and during the 'searching' stage only index is being scanned rather than scanning all documents, hence increase in performance.

The indexer makes an entry in the index for each term or word found in a document and possibly its relative position within the document. Usually the indexer can be configured to ignore stop words. Each search engine uses a proprietary algorithm to create its indices such that, ideally, only meaningful results are returned for each query, in-fact it is a major issue to create and maintain an inverted index when building an efficient search engine.

## 1.2 Expectations from a search engine.

By analysing and studying search patterns and result expectations from user, following are the features which need to be satisfied by a search engine.

- **File formats:** Search engine program should be able to index almost any type of document; It should also support a filter mechanism (to extract text from a file) which is to search word processing documents, pdf files and even various common image formats.
- **Stop-word processing**: Common words, such as "a," "and," and "the," add little value to a search index. But since these words are so common, indexing them will contribute considerably to the indexing time and index size. Search engine program should have an capability for user (programmer) customizable stop word processing.
- **Stemming:** Reducing a word to its root form is called stemming, normally a user expects from a search engine that a query for one word to match other similar words. For example, a query for "select" should probably also match the words "selected," "selector," or "selections."
- **Data sources:** There are some search engines which are indexing data from database or from multiple documents in single file, such as a ZIP archive. Such feature allows programmer to create document index to the indexer through Input Stream or String or even RDBMS tables.
- **Incremental versus batch indexing:** In batch indexing, it is not possible to add document without re-indexing all documents while in incremental indexing, it is easy to add document in the existing index. In some tools, it is very critical to use incremental indexing to handle live data.
- **Indexing control:** Search engines can automatically crawl through a directory and find documents to index. The indexer with crawler provides less flexibility and it requires smooth control over indexed document.
- **Query features:** Different queries used by different search engines like full Boolean queries, only and queries, query which return a "relevance" score with each hit, some query searches only single keywords and there are some queries which searches results from multiple indexes and merge the results to give a meaningful relevance score.
- **Concurrency:** There are some search engines which allows to search from index while the index is simultaneously updating in server.

Presently there are many open source search engines are available in order to make a decision on what search engine to use, it is necessary to complement the results obtained with any additional requirement (i.e. ease of customization). There are some considerations to make, based on the programming language sources and/or the characteristics of the server (e.g. RAM memory available).

Based on our analysis and requirement we found Lucene based search engine to suffice all the requirement mention above, in addition it provides great level of customization.

## 2. Search Engine Tools

### 2.1 What is Lucene?

Lucene is the most flexible and convenient open source full text search engine developed by apache foundation. Integration of Lucene with application is very easy and it controls the behaviour of application. The document model of Lucene is so flexible that it is able to construct and index virtual documents which are having the metadata from the database/XML files.

Lucene supports batch indexing as well as supports efficient incremental indexing so it is possible to add new document as they arrived. The built-in query parser of Lucene supports almost all kind of query features and the search performance is very good. It provides the required search results within less time and gives satisfied quality of the integrated result.

### 2.2 What is SOLR?

As per Apache Solr website Solr is open source enterprise search platform written in java from the Apache Lucene project. It features powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document (e.g., Word, PDF) handling. Solr is highly scalable, providing distributed search and index replication, and it powers the search and navigation features of many of the world's largest internet site.

### 2.3 Solr/Lucene Vs. RDBMS

In traditional relational modelling data is being normalised into interconnected entities to avoid redundancy, for establishing relationship between these entities multiple tables and foreign keys are being used. Whereas Solr/Lucene treat document as collection of fields and forms a document store. Solr/Lucene has provision to accommodate single field containing multiple values; whereas RDBMS approach to do this is joining more than one table by foreign keys.

- Modelling of the fields and their characteristics in tables are required to create database, any single fault may impact on efficiency of RDBMS, but it doesn't happen in Lucene or Solr.
- From a system design viewpoint, two major concerns are clearly partitioned. RDBMS does the record management and searching of records done by Lucene/Solr.The RDBMS has ACID properties while solr has DIH(data import handler) which is used for import data from database.
- Solr and Lucene do not mean to be a replacement for RDBMS. Rather, Solr and Lucene should be used to develop the search service which stores only enough information to efficiently query and given that sufficient information to call object of RDBMS for extra information. The data which are stored in the Solr and Lucene indexes mostly provides a fully searchable view for given set of data.

## 3. An approach to implement Lucene/Solr

Lucene is building multiple index segments and merges them periodically rather than maintaining a single index. Lucene creates a new index segment for each new indexed document, but it merges small segments with larger ones. So the number of segments will be kept small and thus the search will remain fast. If there is a need to update index infrequently then Lucene can merge the segments into one index for fast searching. To prevent the conflicts between index readers and writers, Lucene creates a new segment. At the time of merging, Lucene deletes the old segment and creates a new segment after all readers are closed.

A Lucene index segment contains:

- A dictionary index containing one entry for each 100 entries in the dictionary
- A dictionary containing one entry for each unique word
- A postings file containing an entry for each posting

Segments can be stored in flat files since Lucene never updates them in place. The dictionary index contains offsets into the dictionary file and it holds offsets into the postings file for quick retrieval. Lucene can implements a variety of tricks to compress the dictionary and posting files — thereby reducing disk I/O — without incurring substantial CPU overhead.

### 3.1 Making Lucene/Solr work for Standardise Library Records

In further section of this article we will attempt to explain how we have implemented Lucene/Solr based search engine for library records. Here we have taken MARCXML as our source data.

MARCXML standard is developed by the Library of Congress' Network Development and MARC Standards Office for working with MARC data in a XML environment which is very flexible and extensible to allow users to work with MARC data. Following is a sample of record we have worked with.

**Example:**

```
<?xml version="1.0" encoding="utf-8"?>
<marc:collection xmlns:marc="http://www.loc.gov/MARC21/slim" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://www.loc.gov/MARC21/slim http://
www.loc.gov/standards/marcxml/schema/MARC21slim.xsd">
 <marc:record>
  <marc:leader>00000cam a2200000ua 4500</marc:leader>
  <marc:controlfield tag="001">1</marc:controlfield>
  <marc:controlfield tag="003">INFLIBNET</marc:controlfield>
  <marc:controlfield tag="007">t</marc:controlfield>
  <marc:controlfield tag="008">t1972||||GB||||      00 ||eng </marc:controlfield>
  <marc:datafield tag="016" ind1=" " ind2=" ">
   <marc:subfield code="a">INFL0000000001</marc:subfield>
  </marc:datafield>
  <marc:datafield tag="020" ind1=" " ind2=" ">
  <marc:subfield code="a">333-105133</marc:subfield>
  </marc:datafield>
...
......
  <marc:datafield tag="245" ind1="1" ind2="0">
   <marc:subfield code="a">Philosophy and Linguistics /</marc:subfield>
  </marc:datafield>
  <marc:datafield tag="852" ind1=" " ind2=" ">
   <marc:subfield code="p">1001</marc:subfield>
   <marc:subfield code="p">151836</marc:subfield>
  </marc:datafield>
  <marc:datafield tag="941" ind1=" " ind2=" ">
   <marc:subfield code="h">Gauhati University</marc:subfield>
   <marc:subfield code="s">Assam</marc:subfield>
   <marc:subfield code="c">Guwahati </marc:subfield>
  </marc:datafield>
 </marc:record>
</marc:collection>
```

**3.2 Following is the schematic view system architecture to create a search engine / interface based on MARCXML records described above**

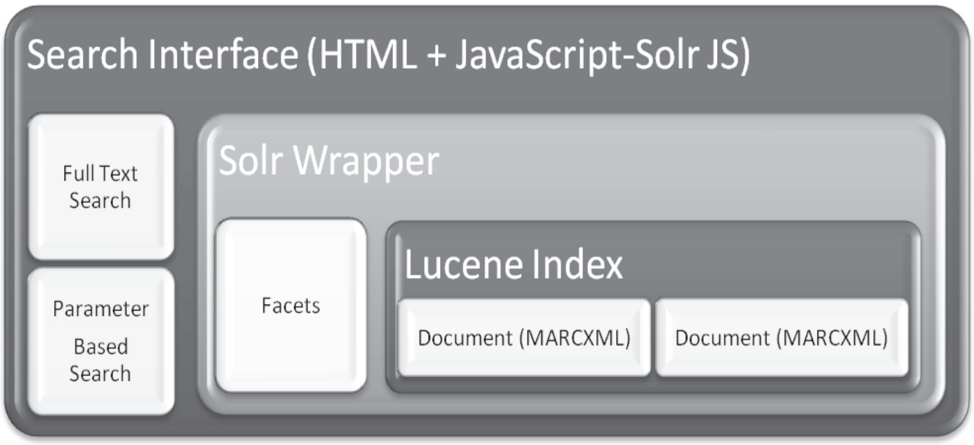


**Figure 1**

## 3.3 Create Index of MARCXML Records

Now the first task would be to create index over MARCXML records. To create index we need to specify what to Index and where to index. To do this there are two files which needs to be modified and configured which are

- solrconfig.xml
- schema.xml

### 3.3.1 Solrconfig.xml

Solrconfig file is most important file that contain parameter for solr configuring itself. It is basically xml file some parameters are given for Solr configuration.

**Datadir**: here we need to specify directory path for hold all index data of MARCXML records.<dataDir>${solr.data.dir:path/to/data}</dataDir>

### 3.3.2 Schema.xml

Schema.xml file have all information about which field you have to index and how those fields dealt when add documents to index. it contain entries like Datatype and Fields.

**Data types**

In schema.xml there is <types> section which allows you to define a list of <fieldtype> declarations. In subclass of FieldType you can use as a field type class, which can be common numeric types (integer, float, etc...) there are multiple implementations provided. one can also define stop words in field type.

**Fields**

In Schema.xml <fields> part is where you can list the individual <field> declaration you can use in your documents. Each <field> has a name that you will use to give field name which you wish to index from MARCXML.

```
<fields>

<field name="id" type="string" indexed="true" stored="true" required="true" omitNorms="true"/>
<field name="Title" type="string" indexed="true" stored="true" omitNorms="true" />

 <field name="Author" type="string" indexed="true" stored="true" multiValued="true"
omitNorms="true" /></fields>

…..

 <field name="AllFields" type="text" indexed="true" stored="true" multiValued="true"
omitNorms="true" />
```

As per Solr documentations field definitions can have following options.

- **name**: mandatory - the name for the field
- **type**: mandatory - the name of a previously defined type from the <types> section
- **indexed**: true if this field should be indexed (searchable or sortable)
- **stored**: true if this field should be retrievable
- **compressed**: [false] if this field should be stored using gzip compression (this will only apply if the field type is compressible; among the standard field types, only TextField and StrField are compressible)
- **multiValued:** true if this field may contain multiple values per document
- **omitNorms**: (expert) set to true to omit the norms associated with this field (this disables length normalization and index-time boosting for the field, and saves some memory). Only full-text fields or fields that need an index-time boost need norms.
- **termVectors:** [false] set to true to store the term vector for a given field. When using MoreLikeThis, fields used for similarity should be stored for best performance.

**Unique key**

The <uniqueKey> of schema.xml file of solr will inform that this field will be unique for all documents, which can be used as handle for each record for manipulation.

```
<uniqueKey>id</uniqueKey>
```

**Default search filed**

The <defaultSearchField> is used by solr when parsing solr query to identify which field name should be searched in QueryParser to use when an explicit fieldname is absent

```
<defaultSearchField>AllFields</defaultSearchField>
```

**Copy fields**

copyField commands is use for copy one field to another at the time a document is added to the index. It's used either to index the same field differently, or to add multiple fields to the same field for easier/faster searching.

Example:

```
<copyField source="Publisher" dest="AllFields"/>
<copyField source="Title" dest="AllFields"/>
```

As shown in above code snippet fields "Publisher" and "Title" will be added to "AllFields" so if user performs general search using "AllFields" matching terms which are part of "Title" or "Publisher" will be available.

### 3.4 Indexing Process

After solr configuration and schema modification, next task would be creating index of available MARCXML records as specified by <dirdata> of solrconfig.xml as explained above. solr will create index all those fields which are specify in schema.xml file. Now create index of all the files available, one need to execute following code snippet for all the documents specified.

```
SolrServer server = new CommonsHttpSolrServer("http://solrserver:port/solr");
Random rand = new Random();
Collection<SolrInputDocument> docs = new HashSet<SolrInputDocument>();
SolrInputDocument doc = new SolrInputDocument();
doc.addField("id",$uniqueid);
doc.addField("Title", $title);
.
.
UpdateResponse response = server.add(docs);
server.commit();
```

After execution of above routine, index for all the documents will be ready as per your specification.

### 3.5 Search from index

Solr performs search on index via HTTP Get in Select URL with Query string using parameter and display search result in Html page. For example, you can use the "Title" parameter to control which stored fields are returned along with relevancy score. You can search from MARCXML index by any fields which are indexed.

Now suppose you want to have all the records where 'Title' contains 'INFLIBNET' following http request will suffice your requirement

http://solrserver:port/solr/select/?q=Title:INFLIBNET

and the output will be as below :

```
<doc>
....
.....
<arr name="AccessionNo">
 <str>EPUN/U/6/6313</str>
 </arr>
<arr name="Keyword">
 <str>Public Performances</str>
 </arr>
...
...
<str name="Title">Application of solr in INFLIBNET</str>
<str name="id">IN00007240</str>
 </doc>
```

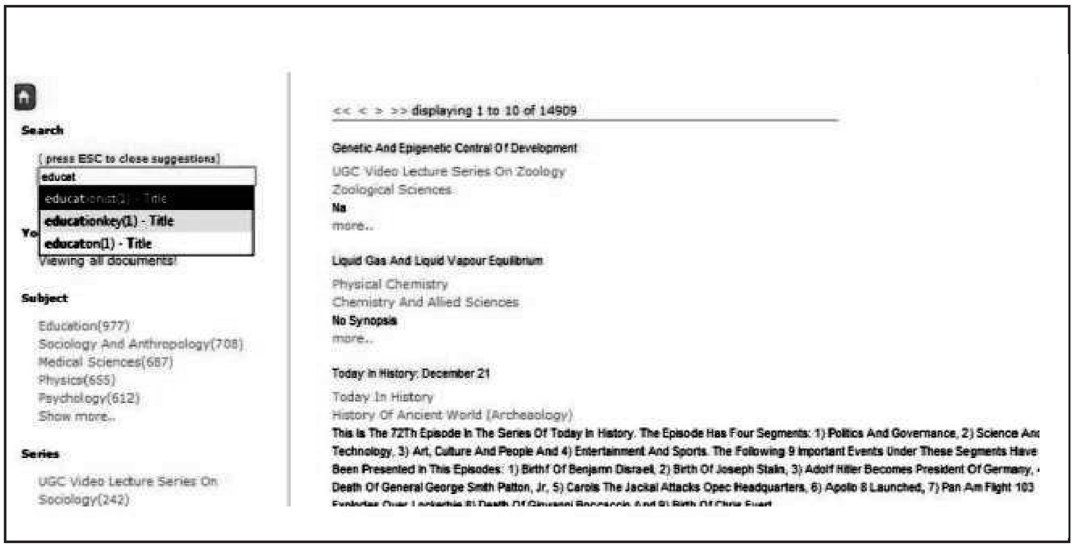
If you want to to perform faceted search which should return facets by keyword, following syntax can be applied
http://localhost:8080/solrdublin/select/?q=Title:INFLIBNET&facet=true&facet.field=Keyword

The facet results will be as shown below at the end of the document :

```
<lst name="facet_counts">
<lst name="facet_queries" />
<lst name="facet_fields">
<lst name="Keyword">
<int name="Economics">43</int>
<int name="Sociology And Anthropology">37</int>
          <int name="Geography And History">35</int>
          <int name="Political Science">20</int>
...
...
...
</lst>
```

The final task would be creating search interface which is basically combination of HTML and javascripts, Here one can used SolrJS which is special javascript library to interact with Solr. SolrJs builds solr query and runs in backend and transformed HTML page (using proper XSLT) will display result of query from index.

To provide parameter based search by which user can search by specific fields like Title,Author etc.User will we need to create search text box and solrjs will dynamically create query according user search input and execute it to solr which in turn examine that query and return search result on html page which is shown in figure. User can search from all those fileds which are mention in field section of schema.xml and indexed.


Search

[ press ESC to close suggestions]

educat

educationist(1) - Title

educationkey(1) - Title

educaton(1) - Title

Viewing all documents!

Subject

Education(977)
Sociology And Anthropology(708)
Medical Sciences(687)
Physics(655)
Psychology(612)
Show more..

Series

UGC Video Lecture Series On Sociology(242)

<< < > >> displaying 1 to 10 of 14909

Genetic And Epigenetic Control Of Development
UGC Video Lecture Series On Zoology
Zoological Sciences
Na
more..

Liquid Gas And Liquid Vapour Equilibrium
Physical Chemistry
Chemistry And Allied Sciences
No Synopsis
more..

Today In History: December 21
Today In History
History Of Ancient World (Archeaology)
This Is The 72Th Episode In The Series Of Today In History. The Episode Has Four Segments: 1) Politics And Governance, 2) Science And Technology, 3) Art, Culture And People And 4) Entertainment And Sports. The Following 9 Important Events Under These Segments Have Been Presented In This Episodes: 1) Birthf Of Benjamn Disraeli, 2) Birth Of Joseph Stalin, 3) Adolf Hitler Becomes President Of Germany, 4) Death Of General George Smith Patton, Jr, 5) Carols The Jackal Attacks Opec Headquarters, 6) Apollo 8 Launched, 7) Pan Am Flight 103 Explodes Over Lockerbie 8) Death Of Giovanni Boccaccio And 9) Birth Of Chris Evert

## 3. Conclusion

Deliberations in paper shows that using Lucene and Solr one can enhance and provide search interface with faceted navigation, result highlighting, fuzzy queries, ranked scoring, This implementation can be customised and optimized for any type of documents including such as text, word, PDF, MARCXML.

## References

1. http://wiki.apache.org/solr/FrontPage
2. http://lucene.apache.org/solr/
3. http://lucene.apache.org/
4. http://wiki.apache.org/solr/SchemaXml
5. http://www.lucidimagination.com/Community/Hear-from-the-Experts/Articles/Solr-RDBMS

## About Authors

**Shri Yatrik Patel,** Scientist –C (CS)INFLIBNET CentreAhmedabad
**E-mail:** yatrik@inflibnet.ac.in

**Shri Dinesh Rayka,** Project Officer (CS)INFLIBNET CentreAhmedabad
**E-mail:** rayka@inflibnet.ac.in

**Shri Divyakant Vaghela,** Project Officer (CS)INFLIBNET CentreAhmedabad
**E-mail:** divyakant@inflibnet.ac.in