Experimental Features

XTF is in constant development, and from time to time experimental features are added. These features are under evaluation, to see if they're designed well and serve their purpose in the most effective manner possible. Rough documentation is provided here for those curious to play with new and possibly very raw code. These features are subject to change, or even to complete removal from XTF. Still, they provide a glimpse into the future of XTF.

Dynamic FRBR

In a catalog with millions of bibliographic records, the problem of duplicate or near-duplicate records often arises. A recognized approach to dealing with this problem is to group them together using Functional Requirements for Bibliographic Records (FRBR).

To implement FRBR in XTF, we chose to adapt the standard FRBR Work Set Algorithm to our purposes. In particular, we changed the method to dynamically determine work groups, rather than making this determination at index time. This allowed us to play with and tweak the algorithm without having to re-index the entire test collection (over 10 million records).

For speed, XTF's "dynamic FRBRization" is implemented in Java, and caches large tables of data drawn from the underlying Lucene index files. In this section we'll cover very briefly the algorithm used, and then show how to activate it.

FRBR Algorithm in Brief

XTF's dynamic FRBR algorithm takes the entire result set from a query, and checks the resulting documents against each other. The goal is to group similar records together. For instance, two records with the same title and author should end up in the same group.

The actual algorithm relies on a score-based approach that allows partial matches of various sorts. It is too complex to cover here, so for full details the reader is referred to Chapter 5 of the Melvyl Recommender Project **Full Text Extension Report** (PDF).

Note: The algorithm involves loading sizable tables for title, author, date, and ID and caching these tables in RAM. This has two major ramifications: first, the tables can take some time to load the first time dynamic FRBR is accessed; second, make sure to give the servlet container plenty of RAM, especially if the number of records in the collection is very large. This can be adjusted by setting -Xmx flag for the Java command that starts the servlet container.

Activating Dynamic FRBRization

XTF integrates dynamic FRBRization into the normal crossQuery process by exposing the grouped records as a new facet. If you're not familiar with facets, refer to the Faceted Browsing section of the **XTF Programming Guide**.

Just like a normal facet, FRBRization is activated by adding a Facet Query Tag to the Query Tag produced by your Query Parser stylesheet. However, it must be of a special form, as follows.

Dynamic FRBR Facet Tag

This tag specifies that the **Text Engine** should group result documents into FRBR Work Sets using the built-in "dynamic FRBRization" algorithm (above). This tag should appear directly within a Query Tag.

```
<facet field="java:org.cdlib.xtf.textEngine.facet.FRBRGroupData({SortOrder}FieldList)"

   <!-- Other attributes as per normal Facet Query Tag... -->

/>
```

where

SortOrder	is an optional field name to sort the resulting groups by. If preceded with a hyphen, the sort is reversed. If not specified, the groups order will be arbitrary. *Note: The braces are required around the sort order specification.*
FieldList	is a required list of the meta-data fields to use to create FRBR work groups. The engine looks for fields containing the strings "title", "author" or "creator", "date" or "year", and "id" to determine how the field contents are to be incorporated into the groupings. *Note: the FRBR algorithm will only work with non-tokenized fields.*

The results are identical to a normal facet query except that the facet groups are computed dynamically based on the result documents of the query, rather than statically based on values in a particular meta-data field.

MARC Record Parsing

California Digital Library (CDL) has experimented with indexing millions of MARC 21 records using XTF. MARC stands for MAchine Readable Cataloging and was developed by the Library of Congress, Adding MARC support is important for ***XTF*** since many existing library catalogs are available only in MARC format, and XTF can add powerful capabilities (such as relevance-based ranking, spelling correction, similarity queries, integration with full text data, etc.) to these catalogs.

Since MARC records are in a binary (non-XML) format, the ***textIndexer*** has the ability to convert them to XML, and then send each record to one or more pre-filter stylesheets.

One might ask: why not convert all the records to MARCXML once, store those XML files in the filesystem, and then index them? While this does work, it has proven very slow, and also quite wasteful of hard drive space. MARC records on their own are very compact, and so great efficiency is gained by converting them *in memory* to MARCXML, and passing that XML directly to the pre-filter stylesheets for indexing.

To enable this processing, when the **Document Selector Stylesheet** encounters a file containing MARC records, it should output a <file> tag with the format attribute set to MARC. The textIndexer will recognize this, open the file, and convert each record to a separate MARCXML record, and pass each record in turn to the pre-filter stylesheets(s). The conversion from MARC to MARCXML is performed by the **marc4j** library, included with the XTF distribution.

CDL's experience with MARC conversion has been very positive: the conversion process is quite fast and the source records remain compact. To deal with some data corruption problems, XTF now makes every attempt to normalize Unicode characters when possible, and skip invalid characters. Also, if an entire record is corrupted, XTF will attempt to re-synchronize on the next uncorrupted record. In this way, the system is now fairly resilient in the face of minor data corruption.

Note that ***dynaXML*** does not handle MARC record parsing or display. In CDL's system, ***crossQuery*** was used to display the records, drawing its data directly from the index.

SRU Servlet

XTF currently contains an experimental servlet that provides exposes a Zing SRW/SRU interface to an XTF repository. Here's a description taken from the Zing web page:

Executive Summary: SRW/U is a low-barrier solution to information retrieval. The SRW/U protocol uses easily available technologies — XML, SOAP, HTTP, URI — to perform tasks traditionally done using proprietary solutions; it can be carried either via SOAP (SRW) or as a URL (SRU).

SRW/U allows users to search remote databases. A user sends a `searchRetrieve` request which includes a query, and the server responds with a `searchRetrieve` response indicating the number of records that matched the query, possibly along with some of those records formatted according to an XML schema that the user requested.

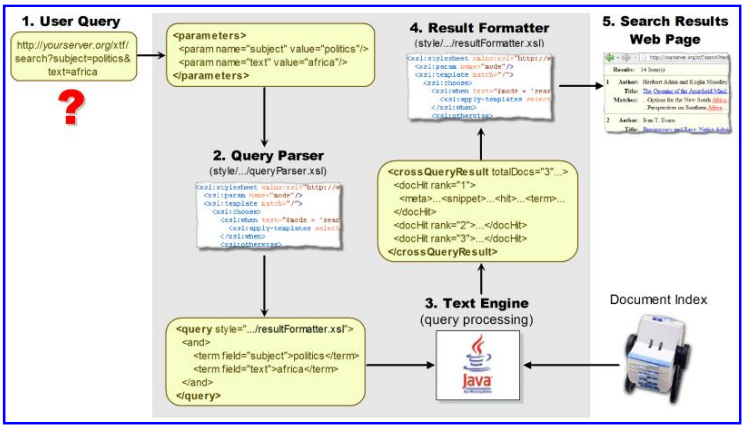
The query is represented in CQL, the "Common Query Language", designed for human readable, human writable, intuitive queries. It supports very simple queries — for example an unqualified single term (e.g. "cat") — but maintains the expressiveness of more complex languages, to represent arbitrarily complex

queries.

The idea here is to make an XTF document repository searchable using the SRU protocol (SRU was simpler to implement than SRW.) This will hopefully allow XTF to play well in a larger world of aggregated repositories and meta-searchers.

Current development plans involve folding the SRU servlet into crossQuery, once support for CQL has been fully integrated. The stylesheets will then recognize SRU queries by their unique URL parameters and redirect to special query parsing and formatting.

As with everything else in XTF, SRU support is provided through cooperation between a Java servlet and several XSLT stylesheets. The SRU servlet operation is almost identical to ***crossQuery***; refer to the diagram below.



Here are the important ways that SRU servlet operation differs from crossQuery:

1. The incoming query URL should obey SRU specifications, and in particular must specify a query in CQL syntax. Typically the URL will also specify an SRU version and an operation to perform (the default operation is `searchRetrieve`.) Here's a sample URL that talks to the SRU servlet:

   `http://`*`yourserver`*`:8080/xtf/SRU?operation=searchRetrieve;query=dc.title=apartheid`
   (Of course, you should adjust the port number and server name according to your own installed servlet container.)

1. The SRU servlet uses a different set of stylesheets. It has its own **Query Parser** and **Result Formatter** stylesheets, located in the `style/SRU` subdirectory of the XTF installation. Also, the servlet has its own configuration file, found here: `conf/sru.conf`
2. The input to the SRU **Query Parser** stylesheet is slightly different. The parameters are all tokenized as normal, but additional parsing is performed on the query parameter: it is parsed as a CQL query. This results in an XCQL query (that is, XML formatted CQL), and this is added to the `<parameter>` block for the query parameter. The query parser does the work of transforming XCQL to a valid XTF query.
3. The output of the SRU **Result Formatter** stylesheet isn't an HTML web page, but rather an XML search result, obeying the SRW specification for results.

Here's some sample output from the SRU servlet:

```
<?xml version="1.0" encoding="UTF-8"?>
<srw:searchRetrieveResponse xmlns:srw_dc="info:srw/schema/1/dc-schema"
                            xmlns:xsi="http://www.w3.org/2001/XMLSchema"
                            xmlns:dc="http://purl.org/dc/elements/1.1/"
                            xmlns:srw="http://www.loc.gov/zing/srw/">

   <srw:version>1.1</srw:version>
   <srw:numberOfRecords>1</srw:numberOfRecords>
   <srw:records>
      <srw:record>
         <srw:recordPacking>XML</srw:recordPacking>

         <srw:recordSchema>info:srw/schema/1/dc-v1.1</srw:recordSchema>
         <srw:recordData>
            <srw_dc:dc xsi:schemaLocation="info:srw/schema/1/dc-schema http://www.loc.gov/z3950/agency/zing/srw/dc-schema.xsd">
               <dc:title>The Opening of the Apartheid Mind: Options for the New South Africa</dc:title>
               <dc:creator>Heribert Adam and Kogila Moodley</dc:creator>

               <dc:subject>African Studies</dc:subject>
               <dc:subject>Politics</dc:subject>
               <dc:subject>African History</dc:subject>
               <dc:description>

                  Refusing to be governed by what is fashionable or inoffensive, Heribert
                  Adam and Kogila Moodley frankly address the passions and rationalities
                  that drive politics in post-apartheid South Africa...
               </dc:description>
               <dc:date>6/28/1993</dc:date>
               <dc:type>text</dc:type>
               <dc:identifier>http://ark.cdlib.org/ark:/13030/ft958009mm</dc:identifier>

               <dc:relation>http://www.ucpress.edu/</dc:relation>
               <dc:relation>http://escholarship.cdlib.org/</dc:relation>
               <dc:rights>Public</dc:rights>
            </srw_dc:dc>
```

```
         </srw:recordData>
      </srw:record>
   </srw:records>
   <srw:echoedSearchRetrieveRequest>
      <srw:version>1.1</srw:version>

      <srw:query>dc.title=apartheid</srw:query>
      <srw:startRecord>1</srw:startRecord>
      <srw:maximumRecords>20</srw:maximumRecords>
      <srw:recordPacking>xml</srw:recordPacking>

      <srw:recordSchema>dc</srw:recordSchema>
   </srw:echoedSearchRetrieveRequest>
</srw:searchRetrieveResponse>
```

The servlet query parser knows how to respond to the `explain` and `searchRetrieve` *operation*s. It supports only *version* 1.1 of the SRU protocol, and requires *recordPacking* to be xml and *recordSchema* to be dc (these are the defaults, so they are optional in the URL.) Other values will produce an appropriate SRU error result.

For further information, you may peruse the stylesheets and the SRW/SRU web page. Comments and improvements are welcome.

Boost Sets

crossQuery includes a feature that allows a **Boost Set** to be specified in the query produced by the **Query Parser**. This set specifies, for each document, a boost factor to be applied to that document. This allows experimentation with different algorithms that assign a global factor to each document (for instance, Google's PageRank algorithm assigns just such a factor.) A subsequent query could specify a completely different boost set, allowing quick side-by-side testing of various global ranking algorithms.

This feature should be considered very experimental, and may be removed at some future time.

A boost set is specified by a `boostSet="`*boostFilePath*`"` attribute to the top-level `<query>` element produced by the **Query Parser** stylesheet. The attribute value should specify a path, relative to the XTF base directory, of a boost set file. Additionally you must specify which meta-data field the keys in the given file should match by adding a `boostSetField="`*fieldName*`"` attribute to the query.

The format of a boost set file is very simple: it should consist of one text line per document, and each line should contain a value from the meta-data field specified by `boostSetField`, followed by a `|` symbol, followed by a factor to be multiplied into the score for that document. For example:

```
doc1|1.5
doc2|2.0
doc4|0.722
```

Boost factors greater that 1.0 will increase the ranking of a document; factors between 0.0 and 1.0 will decrease the ranking of the document; factors less than 0.0 are not valid.

Boost values are multiplied in to the document's basic score calculated by the **Text Engine**. However, the impact can be subtle. To get a better idea of what is going on, you can turn off *score normalization* by adding `normalizeScores="false"` to the query element generated by your **Query Parser** stylesheet. This will turn off the default behavior which is to scale all the scores so that the top document receives a score of 100.

The lines in the file must be listed in ascending order by value. Each value in the file will be matched to an entry in the index. Any documents not matched in the file are considered to have a boost factor of 1.0 (that is, their scores are unaltered.)

Warnings will be logged if values in the file cannot be matched to index entries, and also if any lines are out of order in the file.

Note that if the boost file is very large, it may take some time to read and process the file the first time it is used, but subsequent accesses will be very fast as the result is cached in memory by the ***crossQuery*** servlet.

Sub-document Querying

I figured out a way to add sub-document querying to XTF. It was a bit involved to get it all working, but I'm pretty happy with the result and I don't think I've broken anything.

Several times over the years we've encountered projects involving complex manuscripts, e.g. medieval manuscripts, collected volumes of poetry, etc. There hasn't been a great way to search and display these in XTF. On the one hand, the user probably wants to search by individual poem. On the other hand, when they arrive at that poem it would be good to show it to them in the context of the whole volume. Sub-document Querying is an attempt to address this.

I just checked in [Feb 6, 2009] an experimental implementation which you can play with by grabbing the latest code from SourceForge CVS.

At index time what you can now put `xtf:subDocument="`*mySubDocName*`"` attributes on nodes using your prefilter. The indexer will recognize these and make separate searchable groupings in the Lucene index. Each sub-document inherits meta-data from its parent document. It may also specify its own additional meta-data in the usual way (using xtf:meta attributes).

In ***crossQuery***, your results will have a separate `<docHit>` element for each sub-document matched, and XTF will add a `subDocument="`*mySubDocName*`"` attribute to the `<docHit>` telling you which sub-doc was hit.

In ***dynaXML***, XTF will add a `subDocument` attribute to the top-level `<xtf:snippet>` elements as well.

To restrict full-text search to a particular sub-document only, there is a new query element called `<subDocument>` with the same syntax and usage as the existing `<sectionType>` element (i.e. it can only appear within a query on the 'text' field). This should be particularly useful in dynaXML when jumping from

a ***crossQuery*** hit by rank number, as without this the number wouldn't match up.

Caveat: It is possible to create multiple sub-documents of the same name. It is also possible to have text outside of a sub-document. In these cases, each contiguous section will be a separate search unit, which might not be what the end-user expects. Unless you really want this behavior, care should be taken to put all text with sub-documents and give each sub-document a unique name.

I added a test to my regression suite (under `regress/CrossQuery/0-SubDoc`) so I know things are working, but I haven't made any sample data or stylesheets that use the new feature. I was figuring I'd leave that up to you all.

Edit this entry.

Latest XTF News

- XTF 3.0 beta
- XTF Website Launched
- XTF Community Preview
- XTF 2.2 released

Subscribe to XTF News

- RSS

The **eXtensible Text Framework (XTF)** is supported by the California Digital Library