XQuery Full-Text for the impatient

Xavier Franc

Copyright © Xavier Franc, Axyana Software - 2008

This article is published under the Creative Commons "Attribution-Share Alike" license.

Table of Contents

XQuery and XPath Full Text 1.0 (abbreviated XQFT hereafter) is a new Recommendation (standard) from the W3C that extend the XQuery language with comprehensive functionalities for full-text search.

After a short presentation of main concepts, the present document simply introduces the main features through concrete examples (problem and solution). A more detailed explanation of syntax and semantics would be lengthy and beyond the scope of this document. We would be quite satisfied if this document helps you grasping the essential ideas, and writing your first XQuery Full-Text queries.

1. How XQuery Full-Text is different from usual full-text

- Basically, a standard full-text engine searches a collection of *pages* or *documents* matching a particular query (that is, containing certain words or combinations of words). The engine returns the matching *pages* (or a reference to these pages).

- Standard full-text engines mostly ignore the *structure* of documents, whether this structure corresponds with the formats XML, HTML, PDF, RTF, etc. Roughly speaking, they treat each document as a simple sequence of words.

- Yet some full-text engines are able to divide document contents in areas or *fields*, and use these fields in queries. For example in Google™, there are fields like "page title", "page body", "page links", so for example it is possible search only the pages whose title contains the phrase "yes we can". The Lucene open-source engine allows you to define fields at will.

  So fields are a step towards structuring document contents. As we will see, XQuery Full-Text goes much farther in this direction.

- XQuery deals with XML documents. An XML document is structured in *elements*, which can contain other elements and text (that is, words). XQuery is able to return not only *documents* but precise *nodes* (matching a query) inside documents.

  So naturally XQuery *Full-Text* is able to use XML elements to restrict or refine queries (much like fields in usual full-text engines). This is sometimes called contextual full-text search. For example, in a DocBook document one can search a `section` whose `title` contains the word "enhancement": this is much more selective than searching for a document that contains that word. Here is how it looks:

```
  //section[ title ftcontains "enhancement" ]
```

The great advantage of XQuery Full-Text is that *any element* can be used as a context to refine queries, not only manually designed fields. And of course context elements can be selected with all the power of XQuery expressions.

2. Data used in this tutorial

We will use documents with a relatively simple structure: Shakespeare's plays put in XML format by Jon Bosak. This is a collection of 37 documents which can be found at http://xml.coverpages.org/bosakShakespeare200.html. The top element is PLAY, containing ACT, SCENE, SPEECH and a few others. An element SPEECH is used for each utterance by a character, containing a SPEAKER element for the name of the character, and as many LINE elements as there are lines of text.

Excerpt:

```
<PLAY>
<TITLE>A Midsummer Night's Dream</TITLE>
...
<ACT><TITLE>ACT II</TITLE>
 <SCENE><TITLE>SCENE I.  A wood near Athens.</TITLE>
  <STAGEDIR>Enter, from opposite sides, a Fairy, and PUCK</STAGEDIR>
  <SPEECH>
    <SPEAKER>PUCK</SPEAKER>
   <LINE>How now, spirit! whither wander you?</LINE>
  </SPEECH>

  <SPEECH>
    <SPEAKER>Fairy</SPEAKER>
   <LINE>Over hill, over dale,</LINE>
   <LINE>Thorough bush, thorough brier,</LINE>
   <LINE>Over park, over pale,</LINE>
   <LINE>Thorough flood, thorough fire,</LINE>
   <LINE>I do wander everywhere,</LINE>
   <LINE>Swifter than the moon's sphere;</LINE>
...
```

Note

The way these sample documents are actually accessed is beyond the scope of this document and may depend on the particular XQuery implementation being used.

3. Basic syntax

The fundamental full-text operator is noted by keyword **ftcontains**:

```
   domain ftcontains full-text-query
```

On its right side **ftcontains** requires a full-text query ("full-text selection" in the specifications).

On the left-side, an expression specifies a *search domain*. It should yield a node or generally a sequence of nodes.

The **ftcontains** operator returns a boolean value: true if the full-text query is matched by at least one node in the left-side expression (search domain), false if no match.

In most cases, ftcontains is used as a *predicate* (between square brackets) following a *path expression*. Example (where the path expression is //PLAY):

```
//PLAY[ . ftcontains "juliet" ]
```

This query means: find elements PLAY which themselves (the dot is a shorthand for 'self') contain the word "Juliet".

The search domain is frequently the dot expression '.' (meaning self), but it can be more specific. For example:

```
//PLAY[ TITLE ftcontains "Henry" ]
```

This query means: find elements PLAY whose child element TITLE contains the word "henry" (not case sensitive).

4. Simple queries

Find plays which contain the phrase "to be or not to be":

Solution in XQFT:

```
//PLAY[ . ftcontains "to be or not to be" ]
```

or equivalently:

```
//PLAY[ . ftcontains "to be or not to be" phrase ]
```

```
//PLAY[ . ftcontains { "to be", "or", "not to be" } phrase ]
```

Results:

```
<PLAY><TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
...
</PLAY>
```

Notes:

- A sequence of words (like "to be or not to be") without other specification is a phrase. It is matched by the same sequence of words, in order and without interspersed words.
- A phrase, like any other full-text query can span XML elements. For example:

  ```
   //SPEECH[ . ftcontains  "discontent made glorious" ]
  ```

  This query would return the following result:

  ```
  <SPEECH><SPEAKER>GLOUCESTER</SPEAKER>
  <LINE>Now is the winter of our discontent</LINE>
  <LINE>Made glorious summer by this sun of York;</LINE>
  ...
  ```

Find LINE elements which contain both words "romeo" and "Juliet":

Solution:

```
//LINE[ . ftcontains "romeo juliet " all words ]
```

or equivalently:

```
//LINE[ . ftcontains { "romeo", " juliet" } all words ]
```

```
//LINE[ . ftcontains "romeo " ftand "juliet" ]
```

Notice the keyword `ftand` used to avoid syntax ambiguities with the plain `and`.

Results:

```
<LINE>Is father, mother, Tybalt, Romeo, Juliet,</LINE>
```

```
<LINE>And Romeo dead; and Juliet, dead before,</LINE>
```

```
<LINE>Romeo, there dead, was husband to that Juliet;</LINE>
```

```
<LINE>Than this of Juliet and her Romeo.</LINE>
```

Find LINE elements which contain both words "romeo" and "Juliet" in this order:

Solution:

```
//LINE[ . ftcontains "romeo juliet" all words ordered ]
```

Results:

```
<LINE>Is father, mother, Tybalt, Romeo, Juliet,</LINE>
<LINE>And Romeo dead; and Juliet, dead before,</LINE>
<LINE>Romeo, there dead, was husband to that Juliet;</LINE>
```

Notice that the fourth item of previous query does not match this query because words "romeo" and "Juliet" are not in the required order.

Find LINE elements which contain word "romeo" or word "Juliet" or both:

Solution:

```
//LINE[ . ftcontains "romeo juliet" any word ]
```

or equivalently:

```
//LINE[ . ftcontains { "romeo", "juliet" } any word ]
//LINE[ . ftcontains "romeo" ftor "juliet" ]
```

Notice the keyword `ftor` used to avoid syntax ambiguities with the plain `or`.

Results: 165 occurrences.

Find LINE elements which contain word "romeo" but not word "Juliet":

Solution:

```
//LINE[ . ftcontains "romeo" ftand ftnot "juliet" ]
```

Results: 116 occurrences.

5. Scoring

Getting scores from a XQuery full-text query is done through an extension of the FLWOR (aka Flower) loop:

```
for $hit score $score in //SPEECH[ . ftcontains "king" ]
   order by $score descending
return $hit
```

The keyword **score** introduces a variable that receives the score value. This value is guaranteed to be between 0 and 1, and of course a higher value means a more relevant hit.

The loop above is a typical way of obtaining the results of the query sorted by decreasing score.

How are scores computed ? The answer from the W3C standard is: this is "*implementation-dependent*"... It is very likely however than an actual implementation will take into account the frequencies of query terms in the queried collection.

If several **ftcontains** appear in the expression after the **in** keyword, which one is used to compute the scores ? Again the standard says "*implementation-dependent*". It might the average, the maximum or the first of the score values for each **ftcontains**.

6. Case sensitivity and other matching options

By default, the letter case is not taken into account for full-text search. But XQFT provides several matching options: case sensitivity, diacritics sensitivity (accents), stemming and wildcards:

Find plays which contain the phrase "the King" with this case:

Solution:

```
//PLAY[ . ftcontains "the King" case sensitive ]
```

Find LINE elements which contain the word "Orléans" with its accent:

Solution:

```
//LINE[ . ftcontains "Orléans" diacritics sensitive]
```

Note: this query returns no result as "Orléans" is written without e acute in Shakespeare's plays.

Find LINE elements which contain words matching the pattern ".+let":

Solution:

```
//LINE[ . ftcontains ".+let" with wildcards ]
```

Results: 193, matching Hamlet, Capulet, goblet, doublet etc.

The pattern is a regular expression with limited syntax. Essentially, the dot matches any character and can be followed by occurrence indicators '?', '*', and '+' respectively meaning "optional", "any number" and "at least one".

Find LINE elements which contain word "hammer" or related words by stemming:

Solution:

```
//LINE[ . ftcontains "hammer" with stemming language "en" ]
```

This query would find lines with words "hammer", "hammers", "hammered" etc depending on the capabilities of the stemmer in use.

Reminder: "*stemming*" means reducing a word to a "stem" or radix, and replacing it with a OR of all the words that have the same stem. This is a language-dependent capability, this is why the keyword `language` has to be used jointly.

7. More advanced features

XQFT has a few more advanced features that we will just mention here:

Occurrence counting: find SPEECH elements where the word "love" appears 7 times or more:

Solution:

```
//SPEECH[ . ftcontains "love" occurs at least 7 times ]
```

8 results.

There other occurrence count options: "at most 7 times", "exactly 7 times", "from 6 to 8 times".

Proximity: find LINE elements which contain word1 "near" word2:

Solution:

```
//LINE[ . ftcontains "love tender" all words distance at most 8 words ]
```

This means that words "love" and "tender" may appear in any order, and the number of words from the first to the last (included) must not exceed 8 words.

We have already seen another way of writing this:

```
//LINE[ . ftcontains "love tender" all words window 8 words ]
```

The "distance" option, like "times", has four variants:

```
//LINE[ . ftcontains "love tender" all words distance at least 8 words ]
```

```
//LINE[ . ftcontains "love tender" all words distance exactly 8 words ]
```

```
//LINE[ . ftcontains "love tender" all words distance from 7 to 9 words ]
```

Anchoring: find LINE elements *starting* with phrase "to be":

Solution:

```
//LINE[ . ftcontains "to be" at start ]
```

This means that the first two words inside the matched LINE elements are "to" and "be".

It is also possible to specify "at end", or "entire content" which means both "at start" and "at end".

```
//LINE[ . ftcontains "to be" at end ]
```

```
//LINE[ . ftcontains "to be" entire content ]
```

Mild-not: find LINE elements containing the word "king" but not inside the phrase "king of France":

Solution:

```
//LINE[ . ftcontains "king" not in "king of France"]
```

Note: this query does not mean rejecting the phrase "king of France" always; If the word "king" appears outside of this phrase, a LINE would match, for example this one:

```
    <LINE>No king of England, if not king of France.</LINE>
```

Other features

There are still other features (thesaurus, stop-words, scope, ignored content) that the keen student will find in the W3C specifications.

8. Conclusion

The intent of this tutorial is to provide an idea of the power of XQuery Full-Text:

- XQFT offers a rich set of features, it it probably more comprehensive than the query language of most existing full-text systems.
- It is well integrated, fully composable with the plain XQuery language.
- It allows querying nodes, not only complete documents, and allows using any element as a search context.
- On the minus side, its syntax is a bit verbose and redundant, not very elegant. A number of features are "implementation-defined", which is not optimal for interoperability.