Part VII

Information Retrieval on XML Documents

Outline of this part

1. Information Retrieval and XML
   - IR Functionality
   - XML-IR Languages

IR and XML

In this section, we will briefly look at (contents-based) information retrieval in the context of XML documents. IR functionality is also referred to as Full-Text capabilities of XQuery/XPath.

While we have introduced XML as a data format that can easily represent semi-structured data, we have concentrated very much on structure-based querying/processing up to now. Now, we will discuss

- What IR/full-text actually means,
- how such functionality can be expressed within the XQuery/XPath framework, and
- what (some of) the consequences from an implementation point of view are.

Full-text vs. string matching (1)

Full-text query functionality is significantly different from (sub)string matching in that

- Full-text searches for tokens instead of strings.

Example

Substring search with "lease" will produce a match for a news article entitled "Foobar Corporation releases the 20.9 version of . . . ", while full-text search will not.

- Full-text search is supposed to support language-specific matching, which substring search cannot.

Example

Full-text search for "mouse" should match documents talking about "mice", also you want to be able to search for "XQuery" and "Full-text" occurring within a distance of at most 3 words.
### Full-text vs. string matching (2)

- In a string matching query, there will always be two sets of documents: those that contain a match and those that don’t.
- String matching is an **exact match** problem.

### Tokenization (1)

... denotes the process of breaking a character sequence (the raw document) into “meaningful units”.

- “meaningful” is clearly language-dependent
- typical units are *words, punctuation symbols, and spaces*
- *tokens* are those units that are to be used as basic search units
- subsequent tokens in a document need not be separated by spaces or punctuation symbols, they might even overlap
- *phrases* are sequences of any number of tokens

### Tokenization (2)

Tokenization is a necessary prerequisite for

- operating on parts or roots of tokens (*e.g.*, wildcards and stemming)
- working with relative positions of tokens (*e.g.*, proximities)
- identifying sentences and paragraphs (these notions are highly language-dependent!)

Tokenization (*e.g.*, in XQuery Full-text) is often implementation-dependent. It is typically required, though,

- to preserve the **containment hierarchy**: paragraphs contain sentences, which contain tokens, and
- to identify the same tokens in two codewise identical strings.

Whether or not XML tags make good token boundaries depends on the kind of markup: semantic markup, such as `<title>`, may be a good candidate, while presentation markup, such as `<b>` might not.

### Building a dictionary

In our simplified view of IR, once a document is tokenized, the next step in preparing for retrieval is to generate some form of **dictionary**, *i.e.*, a list of tokens appearing in the document, together with information on where in the document the token occurs.

- In the standard case, token occurrences may, *e.g.*, be described by a byte offset from the document’s start.
- In the case of XML documents, token occurrences might be specified as XPath expressions leading to the enclosing element, and possibly also an offset from the element start.

Notice that building a dictionary (or “inverted list”) is typically performed only after **stop word removal and/or stemming** of the tokens.
Indexing & the retrieval process

Once a dictionary of token occurrences has been built, the last preparatory step for efficient IR is to construct an index, i.e., some clever data structure that guides us to the list of occurrences for a given token. Actually, in many cases, the dictionary is organized in such a way that it can serve as an index as well.

The actual retrieval process—given a few query terms—proceeds as follows (in principle, at least):

1. tokenize the query term(s),
2. lookup the query term(s) in the dictionary using the index, giving a list of occurrences per query token,
3. use a scoring and ranking model to compute a combined relevance value for the intended retrieval units (elements, document fractions, or documents)
4. sort answer elements according to descending relevance and present the answer to the user.

Scoring and ranking (1)

... a wide variety of “retrieval models” have been proposed and evaluated in the literature, in several retrieval benchmarks, and, of course, in practise.

Clearly, the most wide-spread model is the so-called “tf-idf” model, where, in a nutshell, a document’s relevance is higher, if

- the term frequency (tf) of a search term is high in the document, and
- the inverse document frequency (idf) of that term is low.

The rational behind this model is for

- tf: the more often a term occurs in a document, the more relevant that document is as a match for the query term;
- idf: the higher the selectivity, i.e., the more uncommon among all documents, the term is, the better.

Scoring and ranking (2)

We will not elaborate further on retrieval models here, except for a few observations:

- In order to assess the relevance of a document w.r.t. a given query, it may make a big difference, where inside the document the query tokens occur, e.g.,
  - deep down in some nested document structure vs. within top- or high-level elements,
  - in special, semantically meaningful markup-elements (such as, `<title>`, `<abstract>`, `<keywords>`) as opposed to within other elements (not to speak about comments or processing instructions).
- Also, term frequencies may be weighted differently, depending on whether a term occurs several times within the same substructure or across the whole document.
- Obviously, scoring and ranking gets even more complex, if the retrieval target is not the whole document, but just some relevant part of it.

Retrieval units

Classical IR on large document collections delivers whole documents as answer elements. In the case of XML documents, there are two obvious immediate possibilities:

- Continue to deliver whole documents.
  - Largely ignore the XML structure (except for tokenization), disregard markup during retrieval.
  - Queries can refer to “pure text contents” only.
  - ⇒ Miss out a lot of chances!
- Try to deliver “tightly fit” parts of documents.
  - If some `<subsection>` contains all the relevant tokens, deliver this part, not the enclosing document.
  - Often, documents are broken up into a huge number of small “documents” and treated as usual.
  - ⇒ Typical problem: exploding number of “documents”.

-
Content-only vs. content-and-structure XML-IR

The first of the above options will not be considered further, as it makes no use to the XML structure at all. The second one will be referred to as content-only XML-IR.

What we're really after, though, is mixed mode, content-and-structure based XML-IR, since this is what XQuery/Full-text is meant for: A user can navigate/restrict/construct XML documents based on wellknown XQuery functionality, and at the same time use full-text search capabilities intermixed with the rest of the (structure-based) queries. Hence, full-text features must be fully integrated and compositional with the other XQuery operators.

Languages for XML full-text retrieval

Offering new functionality requires a proper user interface. We will not discuss the different proposals in detail, at least the following classes can be identified:

- No particular language, "just use a few search terms" (a “Google”-kind of approach)
- Design a new XML-IR-language (possibly borrowing from other IR languages and from XPath)
  - The NEXI language of the INEX initiative is an example (http://inex.is.informatik.uni-duisburg.de/).
  - Extend existing XML query languages (XPath, XQuery)
    - We will concentrate on this line of development, especially the proposal in W3C http://www.w3.org/TR/xpath-full-text-10/.

XQuery and XPath Full-text

The W3C proposal extends the definitions of XPath and XQuery in three ways:

- It adds a new kind of expression called FTContainsExpr.
- It enhances the syntax of FLWOR expressions in XQuery and for expressions in XPath with optional score variables.
- It adds static context declarations for full-text match options to the query prolog.

Additionally, it needs to extend the data model and processing models in various ways, so as to account for the added functionality.
**FTContainsExpr: Examples**

Example (Return the author of each book with a title containing a token with the same root as "dog" and the token "cat".)

```xml
for $b in /books/book
  where $b/title ftcontains ("dog" with stemming) ftand "cat"
return $b/author
```

Example (The same query in XPath Full-text.)

```xml
/books/book[title ftcontains ("dog" with stemming) ftand "cat"]/author
```

Example (Select books where either the title contains the token "dog" and the token "cat" and the content does not contain a token with the same root as "train", or where the title fails to have one of the matching tokens but the content does.)

```xml
/books/book[title ftcontains "dog" ftand "cat" ne
content ftcontains ("train" with stemming)]
```

**Score variables**

Score variables may be added to the `for` and `let` clauses.

Example (Score variable in `for`.)

```xml
for $b score $s
  in /books/book[content ftcontains "web site" ftand "usability"
and .//chapter/title ftcontains "testing"]
return $b
```

The `for` iterator binds the variable `$b` to each returned book, and `$s` to the corresponding score value for that book. We can sort by score value:

Example (Sorting by score value.)

```xml
for $b score $s
  in /books/book[content ftcontains "web site" ftand "usability"]
where $s > 0.5
order by $s descending
return $b
```

**Score values**

Note that scores apply to entire `for` expressions: even if separate `FTContainsExpr` are used to select, there is still just one score for each match.

```xml
for $p score $s in
  /book[title ftcontains "software"]/para[. ftcontains "usability"]
order by $s descending
return $p
```

Multiple score variables can explicitly influence the overall ranking order:

Return paragraphs ranked by book, then by paragraph score within book.

```xml
for $b score $score1 in /book[title ftcontains "software"]
  order by $score1 descending
return
for $p score $score2 in $b/para[. ftcontains "usability"]
  order by $score2 descending
return $p
```

**Score computation (1)**

The remark on the previous slide ("there needs to be one score value for each item in a result sequence") points to an interesting and still rather open challenge:

- How to compute such single score values from a lot of "component scores" found inside a (nested) XML subtree?
- Some projects have put a significant research effort in exactly this issue. See, for example, the TIJAH project that is now integrating with the Pathfinder family [http://dbappl.cs.utwente.nl/pftijah/](http://dbappl.cs.utwente.nl/pftijah/).
- A "Score Region Algebra" has been developed within TIJAH as a general framework allowing the integration of very different kinds of score computation and information retrieval models into a collection of generic interfaces.
24. IR and XML XML-IR languages

Score computation (2)

N.B. Score values are xs:double from [0, 1]; larger values indicate higher relevance. Scores of FTContainsExprs evaluating to false need not necessarily be 0 neither be > 0 for those returning true.

A score variable in a let clause is also bound to the score of the expression evaluation, but one score is determined for the complete result. The let variable may be dropped from the let clause, if the score variable is present.

Syntax

LetClause ::= ( ("let" "$" VarName TypeDeclaration?) | ("let" "score" "$" VarName) ) \\n\":=" ExprSingle\\n| ("," (("$" VarName TypeDeclaration?) | FTScoreVar) \\n\":=" ExprSingle)*

Marc H. Scholl (DBIS, Uni KN) XML and Databases Winter 2007/08 640

Scoring in let

With scoring, the in-expression of the iteration serves a dual purpose: filter, i.e., drive the iteration, and determine the scores. By using scoring in a separate let clause, it is possible to separately specify expressions for filtering and scoring.

Example

for $b in /books/book[.//chapter/title ftcontains "testing"]
let score $s := $b/content ftcontains "web site" ftand "usability"
order by $s descending
return <result score="{$s}">{$b}</result>

This example returns book elements with chapter titles that contain "testing". Along with the book elements scores are returned. These scores, however, reflect whether the book content contains "web site" and "usability".

Marc H. Scholl (DBIS, Uni KN) XML and Databases Winter 2007/08 641

Scoring is a Second Order feature

Notice that score value are not computed after evaluating the argument, rather, the expression ExprSingle assigned to the score variable is passed to the scoring algorithm. The scoring algorithm calculates the score value based on this expression (not on its value).

In that sense, scoring is a second order functionality, since it takes a (first order) expression as argument!

If scoring were first order, we could also think of writing it as let $s := score(FTContainsExpr). However, then it would clearly be applied to the value of the expression (which can only be true or false), hence, only two different score values could be returned.

... for that reason score is not syntactically written as a function.

Adding weights

Weights can be used inside FTContainsExprs to influence the scoring (i.e., to indicate the relative importance of different search tokens).

- Weighting only makes sense when score values are computed.
- Weights are elements of [0.0, 1000.0], they are only meaningful in relation to each other.
- The default weight is 1.0.

Example

for $b in /books/book
let score $s := $b/content ftcontains ("web site" weight 0.5) ftand ("usability" weight 2)
return <result score="{$s}">{$b}</result>

Marc H. Scholl (DBIS, Uni KN) XML and Databases Winter 2007/08 643
Full-text selections (1)

Full-text selections consist of search conditions possibly involving logical operators (FTOr) followed by an arbitrary number of positional filters (FTPosFilter) optionally followed by a “weight” value specified using an appropriate range expression:

Syntax

\[
\text{FTSelection} ::= \text{FTOr} \text{ FTPosFilter* ("weight" RangeExpr)?}
\]

Syntax (Match options)

\[
\text{FTPrimaryWithOptions} ::= \text{FTPrimary} \text{ FTMatchOption*}
\]

FTMatchOption

\[
\begin{align*}
\text{FTLanguageOption} & \quad \text{FTWildCardOption} \\
\text{FTThesaurusOption} & \quad \text{FTStemOption} \\
\text{FTCaseOption} & \quad \text{FTDiacriticsOption} \\
\text{FTStopwordOption} & \quad \text{FTExtensionOption}
\end{align*}
\]

Syntax (Some more excerpts)

\[
\begin{align*}
\text{FTPrimary} & ::= (\text{FTWords} \text{ FTTimes?}) \mid (\text{\{} \text{Expr} \text{\}}) \\
\text{FTWords} & ::= \text{FTWordsValue} \text{ FTAnyallOption?} \\
\text{FTWordsValue} & ::= \text{Literal} \mid \text{\{} \text{Expr} \text{\}} \\
\text{FTAnyallOption} & ::= \text{\{} \text{\} Anyone} \mid \text{\{} \text{\} All} \\
\end{align*}
\]

Syntax (Positional filters)

\[
\begin{align*}
\text{FTPosFilter} ::= \text{FTOrder} \mid \text{FTWindow} \mid \text{FTDistance} \mid \text{FTScope} \mid \text{FTContent} \\
\text{FTOrder} & ::= \text{\"ordered\"} \\
\text{FTWindow} & ::= \text{\"window\" AdditiveExpr FTUnit} \\
\text{FTUnit} & ::= \text{\"words\" | \"sentences\" | \"paragraphs\"} \\
\text{FTDistance} & ::= \text{\"distance\" FTRange FTUnit} \\
\text{FTRange} & ::= (\text{\"exactly\" AdditiveExpr}) \\
& \quad \mid (\text{\"at\" \"least\" AdditiveExpr}) \\
& \quad \mid (\text{\"at\" \"most\" AdditiveExpr}) \\
& \quad \mid (\text{\"from\" AdditiveExpr \"to\" AdditiveExpr}) \\
\text{FTScope} & ::= \text{\"same\" | \"different\"} \text{ FTBigUnit} \\
\text{FTBigUnit} & ::= \text{\"sentence\" | \"paragraph\"} \\
\text{FTContent} & ::= \text{\"at\" \"start\"} \mid \text{\"at\" \"end\"} \mid \text{\"entire\" \"content\"}
\end{align*}
\]

Full-text selections (2)

A selection “A not in B” matches a token sequence that matches A, but not when it is part of B. In contrast, “A \text{ftand ftnot B}” only finds matches, when the token sequence contains A but not B.

Full-text selections (3)

Recall that the grammar rule for FTSelection allows an arbitrary number of positional filters to follow an FTOr. Multiple adjacent positional filters are applied from left to right, i.e., the first filter is applied to the result of the FTOr, the second is applied to the result of that first application, and so on.

Syntax (Cardinality selection)

\[
\text{FTTimes} ::= \text{\"occurs\" FTRange \"times\"}
\]

Example

In the document fragment “very very big”, the FTWords ...

- "very big" has 1 match consisting of the second “very” and “big”.
- \{"very", "big"\} all has 2 matches; one consisting of the first “very” and “big”, and the other containing the second “very” and “big”.
- \{"very", "big"\} any has 3 matches.
Full-text selections (5)

Syntax (Ignore Option)

Matches inside a certain context can be ignored.

\[
\text{FTIgnoreOption ::= "without" "content" UnionExpr}
\]

Example

If 

\[//\text{annotation}\]

is ignored, "Web Usability" will be found 2 times: in the title and the editor element. The 2 occurrences in the 2 annotation elements are ignored. "expert" will not be found, as it appears only in an annotation element.

```
<book>
  <title>Web Usability and Practice</title>
  <author>Montana <annotation> this author is an expert in Web Usability</annotation> Marigold</author>
  <editor>Véra Tudor-Medina on Web <annotation> best editor on Web Usability</annotation> Usability</editor>
</book>
```

XQuery and XPath Full-text semantics

As mentioned before, full-text and "traditional" XQuery functionality must integrate seamlessly in order to allow for full compositionality. The W3C draft defines the so-called "AllMatches" model, together with translations to and from the XQuery data model.

Towards full-text semantics

1. Evaluate within XDM (i.e., to sequence of items)
2. Convert sequence of items to tokenized text of atoms
3. Evaluate to an AllMatches
4. Convert AllMatches to a sequence of items and scores