XQuery Full-Text extensions explained

S. Amer-Yahia
C. Botev
J. Dörre
J. Shanmugasundaram

There has been recent interest in developing XML query languages, such as XPath and XQuery, to tap the vast amount of information represented and stored in Extensible Markup Language (XML). These query languages, however, have focused mainly on querying the structure of XML documents and provide only rudimentary support for querying text content. To fill this void, XQuery Full-Text has been developed as a full-text extension to XQuery (and also XPath, which is a subset of XQuery). Consequently, XQuery Full-Text can be used to seamlessly query over both the structure and the text content of XML documents. This paper explains the design principles behind XQuery Full-Text, describes its evolution, and illustrates its core features with examples. It is intended as a reference that is shorter and more accessible than the current World Wide Web Consortium working draft.

INTRODUCTION

One of the key benefits of Extensible Markup Language (XML) is its ability to represent a mix of structured and unstructured text data. One can find many real XML data repositories that contain such a mix; for example, the IEEE Initiative for the Evaluation of XML (INEX)[1] data collection contains IEEE papers in XML form, including structured information (such as the names of authors, date of publication, sections, subsections, and references) and also unstructured information (such as the text content of the paper). Other examples of such XML repositories are Library of Congress documents in XML,[2] DBLP in XML,[3] SIGMOD Record in XML,[4] and Shakespeare's plays in XML.[5] Furthermore, application domains, such as the field of library science, have a growing need to seamlessly query over both the structured and text parts of XML documents.

Although current XML query languages, such as XPath[6] and XQuery,[7] can express powerful structured queries over XML documents, they can express only a very rudimentary full-text search. For instance, full-text search in XQuery is expressed using the function: `contains ($e, keywords)`, which returns `TRUE` if and only if the XML element bound to the variable `$e` contains the substring `keywords` (see Reference 8 for a precise definition of `contains`). Although this function is sufficient for simple substring matching, it is inadequate for more complex searches. For instance, consider the following example in the W3C** (World Wide Web

©Copyright 2006 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of the paper must be obtained from the Editor. 0018-8670/06/$5.00 © 2006 IBM

Consortium) XPath and XQuery Full-Text Use Cases document.[9]

(Use Case 10.2.8 Q8): *Consider an XML document that contains books. Find the titles and contents of books whose content contains the phrases* "*usability*" *and* "*Web site*" *in that order, in the same paragraph, using stemming if necessary to match the tokens*.

The XQuery `contains` function is obviously too limited to express the above search, which includes phrase matching, order specifications, paragraph scope, and stemming. The `contains` function also cannot express other full-text operations specified in the Use Cases document, such as general Boolean connectives, distance predicates, synonyms, and thesauruses. Finally, the `contains` function cannot score or rank results, such as returning the top 10 results for a given search.

A full-text extension to both XQuery and XPath, called XQuery 1.0 and XPath 2.0 Full-Text[10] (XQFT), is currently being developed at the W3C to address these issues. XQFT enables users to combine full-text queries with regular XQuery/XPath queries. XQFT makes two extensions to XQuery and XPath. First, XQFT introduces `ftcontains`, a new expression that supports a Boolean full-text search by using a set of fully composable full-text operations. Second, XQFT enhances the XQuery and XPath `FLWOR` expression (explained later) to support scoring of full-text expressions and also other XQuery and XPath expressions.

XQFT was designed by the Full-Text Task Force (FTTF), a W3C XML Query Group task force initiated in the fall of 2002. In February 2003, the FTTF published the working draft of the Use Cases document,[9] which contains a set of use case queries, and in May 2003, the FTTF published the Full-Text Requirements document,[11] which specified the requirements for XQFT. The use cases, written both in English and in XQFT, were inspired by the Library of Congress Use Cases.[12] After the publication of these two initial documents, several language proposals were put forth and discussed by the FTTF participants. The TeXQuery proposal[13] was adopted, with changes, and a first draft of XQFT was published in July 2004. Finally, after incorporating internal and public comments, new versions of the use cases and the language documents were published in April, September, and November 2005.[10]

A few implementations of XQFT have already started to emerge. We are aware of two such implementations: GalaTex is a conformant open-source implementation built on top of Galax,[14] an open-source XQuery system, and Quark, an open-source implementation of XQFT.[15]

The remainder of this paper describes XQFT in more detail. We first provide a brief overview of XQuery and outline the requirements for XQFT. We then describe the various aspects of XQFT based on the November 2005 Working Draft. We finally present some concluding remarks.

XQUERY BACKGROUND

This section presents a high-level introduction to the basic concepts of XQuery and XPath, a subset of XQuery. Only those concepts necessary to understand the full-text extensions are covered here. An introduction to the complete functionality of XQuery is beyond the scope of this paper and can be found elsewhere.[16,17] Note that when we introduce XQuery concepts and XQuery expressions in the following, those concepts and expressions are equally valid for XPath, except when explicitly stated otherwise. The following section is adapted from Don Chamberlin's introduction to XQuery,[16] with his permission.

XQuery data model

Formally, the input and output of XQuery are defined in terms of a data model (described in Reference 16). The query data model provides an abstract representation of one or more XML documents or document fragments. The data model is based on the notion of a sequence. A *sequence* is an ordered collection of zero or more items. An *item* may be a node or an atomic value. An *atomic value* is an instance of one of the built-in data types defined by XML Schema, such as strings, integers, decimals, and dates. A *node* conforms to one of seven node kinds—element, attribute, text, document, comment, processing instruction, and namespace. A node may have other nodes as children, thus forming one or more node hierarchies. Sequences may be heterogeneous; that is, they may contain mixtures of various types of nodes and atomic values.

Consider the sample XML document borrowed from Reference 9 and shown in ***Figure 1***. We will use it throughout the paper to illustrate the functionality of the query language. The data model for this

```
<books>

<book number="1">
  <title shortTitle="Improving Web Site Usability">Improving the Usability of a Web Site
    Through Expert Reviews and Usability Testing</title>
  <author>Millicent Marigold</author>
  <author>Montana Marigold</author>
  <editor>Véra Tudor-Medina</editor>
  <content>
    <p>The usability of a Web site is how well the site supports the users in achieving
      specified goals. A Web site should facilitate learning, and enable efficient and
      effective task completion, while propagating few errors.
    </p>
    <note>This book has been approved by the Web Site
        Users Association.
    </note>
  </content>
</book>

...
</books>
```

Figure 1
Sample XML data

sample is given in ***Figure 2***. This is the representation of the input document on which XQuery expressions operate.

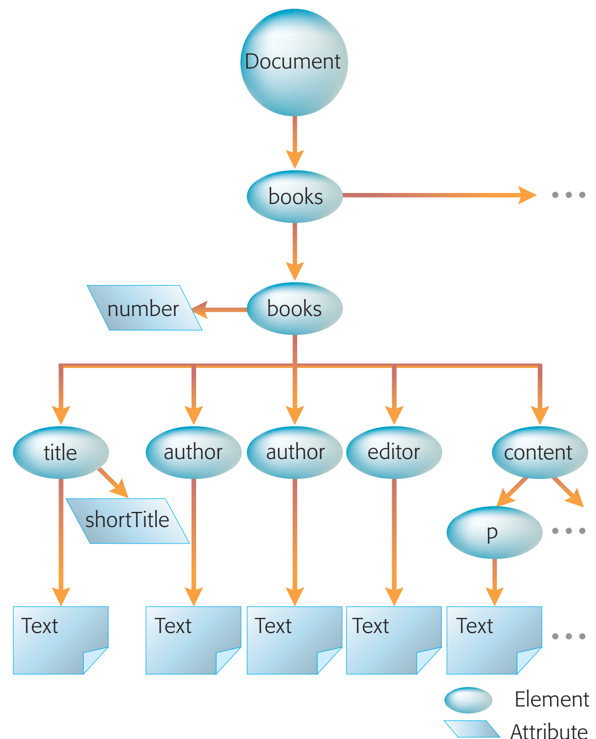

Figure 2
Data model of XML data

XQuery expressions

XQuery is a functional language; that is, it consists of expressions that can be composed by using operators, terms, or function application syntax into more complex expressions. Each expression returns a unique value and has no side effects. The value of a complex expression is determined by first determining the values of the embedded expressions and then applying the composing operator or function call to those values. XQuery expressions are fully composable; therefore, at any place where an expression is expected, any kind of expression may be used.

Literals and variables—Literals and variables are the simplest kinds of expressions. Literals are of one of the primitive types: `integer`, `decimal`, `double`, and `string`. String literals need to be quoted by using either single quotation marks or double quotation marks. `"This is a string"` and `"one more"` are examples of string literals. Numeric literals are written as plain numbers, where a decimal literal is assumed when the number contains a decimal point, and a literal of type `double` is assumed when it contains an exponent; for example, 42 is a very simple XQuery expression representing an integer. A variable in XQuery is a name that begins with a dollar sign. A variable may be used as a placeholder for a value that may be used in several places. We will see how variables are used when we describe iterations in the `FLWOR` expression paragraph.

Function calls—Another basic form of expression is the call to a function. XQuery allows the use of predefined or user-defined functions; for example, the `contains` function mentioned earlier is predefined.

Path expressions—When dealing with hierarchical data like XML, one of the most important capabilities of a query language is to make it very easy to select parts of the hierarchy. XQuery uses *path expressions* for this purpose. A path expression consists of one or more steps, separated by a "/". A step can be thought of as a means to navigate through the hierarchy of nodes of the data model, allowing the selection of certain sets of nodes based on their position in the hierarchy. XQuery supports steps along various axes, the most prominent one being the `child` axis. A step along the `child` axis allows—given a context node—the selection of certain (direct) children nodes of that node. Each step in a sequence of steps takes the result of the previous step as its context. If the previous result is a sequence of nodes, each of them is taken as a context node in turn, and the union over these different evaluations of the step is returned. The following is an example of three steps along the `child` axis:

```
/child::books/child::book[child::author =
                            "Montana Marigold"]
             /child::title
```

Starting with the document node, the top-level node in the hierarchy, the first step selects the unique `books` element. The next step selects from that element all its `book` element children that have some author `child` element with the text *Montana Marigold*. Finally, the last step takes as context all those `book` elements and selects all of their `title` children elements. Those `title` elements are the result of the whole expression.

Apart from the child axis, XQuery supports up to 11 other axes that allow navigation in one step from an element to its attribute nodes, its parent, all of its descendant element nodes, its preceding or following sibling in the hierarchy, and so on. For navigating along the most common axis, XQuery has an abbreviated syntax. The child axis is the default when no axis is specified using "::", allowing the previous example to be written as:

```
/books/book[author = "Montana Marigold"]/title
```

Here are two more examples of path expressions, illustrating the abbreviated syntax. In `$books//note`, all `note` elements that are descendants of the nodes bound to `$books` over one or more levels are selected. The `book/@number` path shows how to select attribute nodes (`@` is simply short-hand for `attribute::`).

Predicates—Predicates are used to further refine the selection of a step expression. A predicate is an expression, written in square brackets, that returns a Boolean value. The predicate expression inside [ ] is evaluated for each of the items resulting from the step that it is refining. Only the items for which the predicate expression evaluates to `TRUE` are retained. The preceding example contains a predicate that filters the sequence of all `book` elements so that the only elements kept are those that have an `author` element with the given text.

All the concepts and expressions introduced so far are common to XQuery and XPath. The following form of expression is an XQuery expression only and hence, is the major difference between XPath and XQuery.

FLWOR expression—`FLWOR` (pronounced as flower)—an expression for iteration, sorting, and variable binding—is one of the most powerful constructs in the XQuery expression language. Its name comes from the keywords of the clauses of which it consists: `for`, `let`, `where`, `order by`, `return`.

The `for` and `let` clauses are used to generate a tuple stream, where each tuple is simply an ordered sequence of the bindings for each of the specified variables. An iteration over the subsequent clauses is performed, using each tuple of the stream in turn. With the optional `where` clause, the tuple stream can be filtered, and with the optional `order by` clause, it can be reordered. Finally, for each remaining tuple, the `return` clause is evaluated, and the result is concatenated to the overall result sequence of the whole expression.

For example, the following `FLWOR` query causes an iteration with the variable `$book` being bound in turn to each of the book elements in the given document under the path `/books/book`. It returns the book numbers of those books that have a date

attribute larger than "2003-12-31" and have more than one author element, sorted by date:

```
for $book in doc("library.xml")/books/book
where $book/@date >
  "2003-12-31" and count ($book/author) > 1
order by $book/@date
return $book/@number
```

As we shall see, the FLWOR expression is also key to integrating the part of the full-text functionality that has to do with scoring (and hence ranking) results.

XQFT REQUIREMENTS AND ALTERNATIVE APPROACHES

In this section, we briefly describe the main requirements that influenced the design of XQuery Full-Text and highlight related work in these areas. For more details, we refer the reader to the FTTF Requirements document.[11] In light of the FTTF requirements, we classify previous approaches based on how well they satisfy the requirements and also discuss the limitation of a function-based approach that was originally considered within the FTTF.

XQFT Requirements

Searching over semistructured data—Users should be able to specify both the search context (the set of nodes over which the full-text search is to be performed) and the return context (the part of the document collection that is to be returned). In traditional full-text search,[19] both the search and return contexts are usually the entire document collection. However, in the case of structured or semistructured XML documents, it is often desirable to search over and return document fragments, for example, searching only on a book abstract rather than an entire book to return the book title and authors.

Expressive power and extensibility—Users should be able to express complex full-text searches, and the language should be extensible. For example, full-text primitives are Boolean connectives, distance predicates, phrase matching, stemming, and thesauruses. Further, users should be able to compose these primitives arbitrarily.

Scores and ranking—Users should be able to obtain relevance scores for the results of full-text searches, control how scores are computed, obtain the top-*K* results, and specify a scoring condition, which is possibly different from the full-text search condition. Many measures, such as TF*IDF (term frequency times inverted document frequency)[19] and term proximity, can be used to obtain the relevance scores.

Integration with XQuery—Users should be able to embed full-text searches in XQuery expressions and vice versa without extending the XQuery data model. This enables users to query seamlessly over both structured data (using XQuery) and full-text data (using full-text search).

Language properties and efficiency—The language syntax should allow for static type checking and inference, and the language should not preclude an efficient implementation.

Overview of prior approaches

Various ranking models have been proposed for XML in the information retrieval (IR) literature. Particularly influential among these approaches are XIRQL[20] and XXL,[21] which extend the probabilistic model,[22,23] and JuruXML[24] and ELIXIR,[25] which extend the vector space model.[26] The INEX initiative[1] was also established to systematically evaluate various ranking methods for XML. These methods have shaped the design of XQFT, but while the focus of these methods is on ranking, XQFT provides a framework in which these ranking methods can be implemented.

> ■ XQFT can be used to seamlessly query over both the structure and the text content of XML documents ■

Several languages have been proposed for processing XML data on structure and text. Some of these solutions explore a few full-text search primitives at a time (e.g., Boolean term retrieval,[27,28] term similarity,[21,25] proximity distance,[22] and relevance ranking[20,22,24,25,29,30]). Other languages, including XIRQL,[20] TIX,[31] and TOSS,[32] extend XQuery with ranking and a few full-text primitives. *Table 1* classifies existing XML full-text search languages and systems according to desired search primitives and scoring methods. From the table, we can see that XQFT fills a gap in the space of expressiveness of query languages for XML. Most of the existing languages include limited XPath navigation in the input query and allow Structured-Query-Language-

Table 1 Classification of existing IR engines for XML

IR Engines	XML Query Engine	Search Primitives	Weighting on Query Terms	Similarity Operator	Scoring
TeXQuery[13] (Quark)[†]	XQuery	Phrase matching, Boolean connectives, order specifier, proximity distance, number of occurrences, match options (stemming, regular expressions, stop words, case sensitive)	Yes	Implicit	Probabilistic or vector-based model
XQuery Full-Text[10] (GalaTex)[†]	XQuery	Phrase matching, Boolean connectives, order specifier, proximity distance, number of occurrences, match options (stemming, regular expressions, stop words, case sensitive)	Yes	Implicit	Probabilistic or vector space model
XIRQL[18] (HyREX)[†]	XQL	Phrase matching, Boolean connectives, `sounds_like` operator	Yes (Query terms and document terms)	Textual and context	Probabilistic model
Flexible XML Search[19] (XXL)[†]	XML-QL	Phrase matching, limited Boolean connectives, `LIKE` operator	No	Textual and context (similarity join)	Probabilistic model
ELIXIR[23]	XML-QL	Phrase matching, limited Boolean connectives	No	Textual (similarity join)	Vector space model
JuruXML[22]	Juru	Phrase matching, limited Boolean connectives (negation)	No	Implicit, textual and context	Vector space model

[†] Language names are followed by system or prototype names in parentheses.

(SQL)-like queries (ELIXIR, XXL, XIRQL). Other languages have considered a more simple and intuitive query syntax by specifying the query either as an XML fragment (JuruXML) or in a Google**-like style through a list of pairs: element name and term (XSearch). There are different approaches to the granularity of query output. XXL and ELIXIR are able to return document fragments. In contrast, XIRQL and JuruXML focus more on relevance-oriented search and let the engine decide which nodes to return. Some existing languages incorporate explicit or implicit textual and context (element names) similarity operators used in the ranking method.

Limitations of extending XQuery with full-text functions

XQuery is a functional language. It is thus natural to think of using functions to extend it with full-text search capabilities. In this section, we describe two different function-based approaches that were considered within the FTTF and discuss their limitations.

In the first approach, we create a new `contains`-like function for each full-text primitive (such as Boolean connectives and distance predicates) and compose these functions to create complex full-text queries. As an example, consider a query that finds all nodes (bound to variable `$n`) that contain the search token "`usability`" and either the search token "`testing`" or the search token "`analysis`". Further, the search terms should be within a window of size 10 (i.e., a window of at most 10 terms should contain all the search terms). This query can be written as follows:

```
distance(contains ($n, "usability")
  and (contains ($n, "testing")
  or contains ($n, "analysis")),10)
```

The main problem with using this approach in the context of XQuery is that it requires an extension of the XQuery data model. This is because the distance function cannot determine if the search terms are within a distance of 10 from each other based solely on Boolean values returned by the `contains` function unless some extra information about search token positions is somehow returned with the Boolean value—this is essentially a fundamental extension to the XQuery data model and violates the requirement for tight integration with existing XQuery expressions.

In the second approach, we extend the `contains` function so that this single function is used to embed all full-text primitives, similar to SQL/MM.[33] SQL/MM extends SQL to express queries on text, images, and spatial data (see also Reference 32 for a related abstract-data-type-based approach). By adopting the same approach, all the processing related to full-text search (including distance-based predicates) is expressed entirely within the `contains` function, and the XQuery data model would not have to be extended. For instance, the preceding example can be written as follows in SQL/MM-like syntax:

```
contains($n, "usability and (testing or analysis)
  distance 10")
```

The main problem with this approach is that the full-text search is specified in an uninterpreted string that is opaque to the rest of the XQuery language. This causes a problem when we wish to embed XQuery within full-text searches, such as in a query that finds all articles that mention the title of one of Richard Dawkin's books. Here the search terms—the words in the titles of Richard Dawkin's books—are themselves the result of an XQuery expression, and there is no natural way to embed these results into the full-text search string, thereby violating the composability of XQuery and Full-Text. One could think of generating the full-text search string during query evaluation by using string concatenation on the results of the XQuery expression as follows:

```
contains ($n, concat ( //book[author =
  "Dawkins"]/title, "and"))
```

However, this implies that full-text search string will not be created until runtime, which means that even simple syntax errors in the string cannot be checked until runtime (such as an `and` operator with only one operand in the preceding example).

XQFT OVERVIEW

The XQFT language, which satisfies all of the requirements outlined in the previous section, makes two extensions to XQuery. First, XQFT introduces a new XQuery expression, called `FTContainsExpr`. The `FTContainsExpr` expression enables users to search over XML nodes with an arbitrary combination of full-text primitives. Because `FTContainsExpr` is an XQuery expression, it also can be arbitrarily composed with other XQuery expressions.

Second, XQFT extends the XQuery `FLWOR` clause to score and rank the results of an `FTContainsExpr` expression. The extension to `FLWOR` can also score XQuery expressions besides `FTContainsExpr`, but a discussion of this capability is beyond the scope of this paper.

In the next two sections, we describe the `FTContainsExpr` expression and the scoring extensions in more detail.

XQFT: FTContainsExpr

The `FTContainsExpr` expression consists of two parts. The first part specifies the sequence of XML nodes over which the full-text search is to be performed. We call this sequence the *search context*. The second part specifies the full-text search condition. The full-text search condition is specified using expressions called `FTSelection`, which express simple term search queries as well as more complex phrase matching, such as Boolean connectives, proximity operators, stemming, and thesauruses.

The `FTContainsExpr` expression has the following syntax:

```
FTContainsExpr ::=
  Expr "ftcontains" FTSelection
```

`Expr` is an XQuery expression that specifies the search context, which is the sequence of XML nodes over which the full-text search is to be performed. (The issue of whether the search context can contain atomic values is still under discussion at the FTTF.) `FTSelection` specifies the full-text search condition.

FTContainsExpr returns a Boolean value that is true if and only if some node in the search context satisfies the full-text search condition.

In order to evaluate a full-text search condition over a search context node, all textual content of that node is (conceptually) transformed into a sequence of words, or more generally, tokens, by a process called tokenization. Those tokens are the units that search predicates ultimately can "look for." In XQFT, the process of tokenization is left to be defined by the implementation, as it is highly language- and domain-dependent. Note, however, that the tokenization process establishes the most fundamental difference between pure substring matching and full-text search. We use "word" and "token" interchangeably, and when we talk about matching a word or a phrase, we generally mean matching a token or a sequence of tokens.

Examples

We now present several examples of queries that use the FTContainsExpr expression:

```
//book ftcontains "web" && "usability"
```

The above query returns TRUE if and only if some book in the search context //book (which is an XQuery expression) contains the search terms *web* and *usability*. Here "web" && "usability" is a simple example of an FTSelection. Note how FTContainsExpr can limit the search context by using an XQuery expression (//book in the above example). Further, because FTContainsExpr returns a result in the XQuery data model (a Boolean atomic value), it can be arbitrarily nested within other XQuery expressions. A more complex example illustrating this aspect is given below (Query 4.2.1 in Reference 9):

■ XQFT is a full-text extension to XQuery (and also XPath, which is a subset of XQuery) ■

```
book[.//@shortTitle ftcontains "improve" &&
  "web" && "usability"]/title
```

The above query returns the titles of those books whose short title contains the search terms *improve*, *web*, and *usability*. Note how the FTContainsExpr expression (.//@shortTitle ftcontains "improve" && "web" && "usability") is nested within the XQuery expression book [...]/title.

There are two other interesting aspects to note about the preceding query. First, the query shows how XQFT can specify a return context, or the part of the selected XML items that are to be returned. Specifically, the return context is only the titles of the selected books (and not the contents of these books). Second, XQFT takes advantage of existing XQuery constructs, such as path expressions, to specify the search context (.//@shortTitle) and the return context (/title).

We can use the composability of the FTContainsExpr expression with XQuery to construct more sophisticated queries with which we can search in multiple search contexts, as illustrated below:

```
//book [(metadata ftcontains "usability tests") and
  (content/part/chapter/title ftcontains
    "web-site usability")] /title
```

The above query returns the titles of books that contain the phrase *usability tests* in their metadata and the phrase *web-site usability* in a chapter title.

Finally, the following example shows how an XQuery expression can be used also inside the FTSelection of an FTContainsExpr expression:

```
//book[.//section ftcontains
  {//article[@id = 10]/title} all words]
```

The above query returns the books that contain at least one section such that the section contains all the words in the title (or titles) of the article with id = 10. Here, the full-text search condition is {//article[@id = 10]/title} all words and is based on the XQuery expression //article[@id = 10]/title. The keywords all words state that all the words in the title (or titles) should be present in the relevant section of the book.

FTSelection expressions

As mentioned above, FTSelection expressions are used to specify the full-text condition in an FTContainsExpr expression. There are many types of FTSelection expressions and they are fully composable, so that users can construct complex

full-text conditions. We now describe the `FTSelection` expressions supported in XQFT.

Word and phrase matching: The simplest `FTSelection` expression contains a single word or phrase in a search string. The following two queries return books that contain the word *usability* and the phrase *usability testing*, respectively:

```
//book[. ftcontains "usability"]
//book[. ftcontains "usability testing"]
```

As mentioned earlier, an `FTSelection` expression can also be the result of an XQuery query (the latter must be enclosed in curly braces {}). The following query returns books that contain an occurrence of one of the section titles of the article with id = 10 matched as a phrase:

```
//book[. ftcontains
  {//article[@id = 10]/section/title} any]
```

For example, if the expression in curly braces evaluates to the sequence (`"site usability"`, `"testing"`), books are required to contain the phrase *site usability* or the word *testing*.

Other possible options are `any word`, `all`, `all word`, and `phrase`. The difference between the `any` and `any word` options is that in the latter case, the elements of the sequence are not matched as phrases, but are tokenized into separate words and searched individually. Therefore, if in the preceding example `any word` would be used instead of `any`, the query would require only that books contain any of the words *site*, *usability*, or *testing*. Likewise, if the `all` option is used on a sequence, then all elements of the sequence are required to be contained simultaneously as phrases (*site usability* and *testing* in our example), whereas the `all word` option requires only that all of the individual words be contained (*site*, *usability*, and *testing* in our example). Finally, the `phrase` option requires that all strings from the sequence returned by the nested XQuery expression be concatenated into a single phrase, which is to be matched. For example, the next query requires that books contain the phrase *site usability testing:*

```
//book[. ftcontains
  {//article[@id = 10]/section/title} phrase]
```

Boolean operators—`FTSelection` expressions can be combined by using Boolean operators to create more complex full-text search conditions: `&&` specifies a conjunction of words, `||` specifies a disjunction, `!` specifies negation or absence of a word, and `not in` (also called *mild negation*) specifies that a word or phrase is not considered a match if occurring in a given context. For example, the following query returns books that contain both the word *usability* and the word *testing*:

```
//book[. ftcontains "usability" && "testing"]
```

Note that the FTSelection expression in the above query (`"usability" && "testing"`) contains two simpler `FTSelection` expressions (`"usability"` and `"testing"`) combined by using a `&&`. As a more complex example, the following query combines the `&&` and `||` operators to return books that contain both *site* and *usability* or both *usability* and *testing*:

```
//book[. ftcontains ("site" && "usability") ||
                    ("usability" && "testing")]
```

The negation `!` specifies that a full-text search condition must not be satisfied in the search context. For example, the following query returns books that contain the phrase *New Mexico* but not the phrase *Mexico City*:

```
//book[. ftcontains "New Mexico" && !"Mexico City"]
```

Sometimes the negation `!` can be too restrictive and can produce unexpected results. As an illustration, consider a user who is interested in books about Mexico, but not about New Mexico. Assume the user expresses the query using `!` as follows:

```
//book[. ftcontains "Mexico" && ! "New Mexico"]
```

The query will not return a book about Mexico if it contains a statement such as *Mexico shares a border with New Mexico*. Clearly, this is not what the user intended. To address this issue, XQFT supports a weaker notion of negation using the `not in FTSelection`. The binary operator `not in` returns nodes from the search context that contain words satisfying the left operand, so long as the same words are not part of a match that satisfies the right operand. For example, the following query will return the books that contain an occurrence of the word *Mexico* that is not part of the phrase *New Mexico:*

```
//book[. ftcontains "Mexico" not in "New Mexico"]
```

Distance predicates—In many applications, users may wish to specify the distance between words in the full-text condition. For example, a user may wish to search for books where the words *site* and *usability* occur close to each other. XQFT supports three ways to specify such distance predicates.

The first approach uses the `same sentence`, `different sentence`, `same paragraph`, and `different paragraph` `FTSelection` operators, which specify that the words should occur in the same sentence, different sentence, same paragraph, or different paragraph, respectively. The boundaries of a sentence and paragraph are determined by an implementation-dependent tokenizer that operates on the search context, as these concepts may vary across languages. Apart from the Boolean operators, all `FTSelection` operators are postfix operators that are appended to an `FTSelection` expression to form a new `FTSelection` expression. For example, the following query returns books that contain the words *site*, *usability*, and *testing* in the same paragraph:

■ Although current XML query languages can express powerful structured queries over XML documents, they can express only a very rudimentary full-text search ■

```
//book[. ftcontains
  ("site" && "usability" && "testing")
    same paragraph]
```

Similarly, the following query returns books that contain the words *site*, *usability*, and *testing* such that each of them appears in a different sentence:

```
//book[. ftcontains
  ("site" && "usability" && "testing")
    different sentence]
```

The second approach to specifying distance predicates is using the `distance` `FTSelection` operator, which specifies the distance between every two consecutive occurrences of the matching words in units of words, sentences, or paragraphs. For example, the following query returns books that contain the words *site*, *web*, and *testing* in the same sentence, so that there is a triple occurrence of these three words where every two consecutive occurrences do not have more than one intervening term:

```
//book[. ftcontains ("site" && "web" && "testing")
  same sentence distance at most 1 words
```

Note that the distance is given as a range (between 0 and 1 in this example). Other options available for specifying the allowable range of distances are `at least E` (denoting the range $[E, +\infty]$), `exactly E` (denoting the range $[E, E]$), and `from` E_1 `to` E_2 (denoting the range $[E_1, E_2]$). Here E, E_1, and E_2 are XQuery expressions that evaluate to an integer number.

The third approach to distance predicates is using the `window` `FTSelection` operator. It specifies that words or paragraphs must be matched (or not matched) within a certain number of consecutive words, sentences, or paragraphs within the text. The following first query returns books that contain the words *site* and *usability* within a window of at most three words, whereas the second query requires an occurrence of *web site* such that neither the sentence preceding it nor the sentence following it contains the word *testing*:

```
//book[. ftcontains ("site" && "usability")
  window 3 words]
//book[. ftcontains ("web site" && ! "testing")
  window 2 sentences]
```

Order of the words—The `ordered` `FTSelection` operator specifies whether the words in the search context should occur in the same order as they appear in the query. For example, the following query returns books that contain the word *site* before the word *usability* within a window of three words:

```
//book[. ftcontains ("site" && "usability")
  ordered window 3 words]
```

Number of occurrences—The `occurs` `FTSelection` operator can be used to specify the number of distinct occurrences of a full-text search condition. For example, the following query returns books that contain at least two distinct instances of occurrences

of the words *site* and *testing* within a window of three words:

```
//book[. ftcontains ("site" && "testing")
  window 3 words occurs at least 2]
```

String content—The `FTContent` operator is used to find matches in which the words and phrases are the first, last, or all of the words and phrases in the tokenized string value of the element being searched. For example,

```
/books//title[. ftcontains
  "improving the usability of a web site" at start]
```

finds title elements starting with the phrase *improving the usability of a web site*. If `at end` was used instead of `at start`, the query would find title elements ending with the phrase *improving the usability of a web site*. Finally, the `entire content` option would return title elements where the phrase *improving the usability of a web site* constitutes the entire content of the title.

Match options

Although `FTSelection` expressions are used to find search context nodes that contain exact matches for the query words, in many cases, users may also be interested in context nodes that do not contain exact matches for the query words, but contain similar matches. For example, a user searching for search context nodes that contain the word *usability* may also be interested in search context nodes that contain the word *usage* (with the same stem as *usability*, namely *use*), or the word *Usability* (with the same spelling as *usability*, but with an uppercase character), or the word *easy-to-use* (with the same semantic meaning as *usability*). Match options are used to specify such relaxations on the query words so that they can be matched in a more flexible manner with the search context nodes.

Match options can be seamlessly composed with `FTSelection` expressions. A match option applied on a (possibly complex) `FTSelection` expression applies to all query words and distance predicates within the `FTSelection` expression. We now describe the match options supported in XQFT.

Stemming—Implementations of full-text search usually have a means to extend the result set by looking for linguistic variants of the query terms, such as *use*, *used*, and *using*. In XQFT, the "with stemming" match option is used for this purpose, and the "without stemming" match option is used to disable this feature. For example, the following query returns books that contain the word *achieve* as well as all words that share the same stem as *achieve* (such as *achieving*):

```
//book[. ftcontains "achieve" with stemming]
```

Stemming can also be selectively disabled, as illustrated in the following query that returns books which contain the word *Tudor-Medina* without applying stemming, and contain the words *site* and *testing* in the same sentence, using stemming to match *site* and *testing*:

```
//book[. ftcontains
  ("Tudor-Medina" without stemming &&
    ("site" && "testing" same sentence))
      with stemming]
```

Note that the outermost "with stemming" applies to the entire `FTSelection`, except where it is explicitly overridden within (for *Tudor-Medina*).

The exact method used to perform stemming is implementation defined. Hence, implementations are free to provide more sophisticated linguistic matching than a simple stemming approach, which in many languages gives poor results.

Character case variations—The `case sensitive`, `case insensitive`, `lowercase`, and `uppercase` match options deal with variations in the character case of words. By default, the `case insensitive` match option is used, which means that the case of the words is not considered when interpreting the full-text search condition. For example, the following two queries are equivalent and will return books that contain the words *Usability* and *testing*, ignoring the case of the words:

```
//book[. ftcontains "Usability" && "testing"]
//book[. ftcontains ("usability" && "testing")
  case insensitive]
```

Although the `case insensitive` match option is the default, users may wish to explicitly specify it because the default can be overridden in the query prologue or by using another case match option at a higher level of the query.

The `case sensitive` match option is used to match the search context nodes that contain exactly the same word (in the same case) as the query. For example, the following query returns books that contain the word *LaTeX* in which each character of the word is spelled in exactly the same way:

```
//book[. ftcontains "LaTeX" case sensitive]
```

The `lowercase` and `uppercase` match options match words that appear as all lowercase or all uppercase in the search context. For example, the following query returns books that contain all the words from each title of the article with id = 10, with all the words being interpreted as uppercase words:

```
//book[. ftcontains {//article[@id = 10]/title}
  all words uppercase]
```

Diacritics—The `diacritics sensitive`, `diacritics insensitive`, `with diacritics`, and `without diacritics` match options deal with diacritical marks in characters, such as accents, diaeresis, and cedillas (see the Unicode standard[35] for a definition of diacritical marks). By default, the `diacritics insensitive` match option is used, which means that when matching a word in a search context node against a query word, diacritical marks are ignored. For example, the following three queries return the same set of books—books that contain the word *Vera*, possibly including diacritics:

```
//book[. ftcontains "Véra"]
//book[. ftcontains "Véra" diacritics insensitive]
//book[. ftcontains "Vera" diacritics insensitive]
```

The `diacritics sensitive` match option requires words in the search context nodes to contain exactly the same characters as the respective query words, including diacritics. The `with diacritics` and `without diacritics` match options both imply a `diacritics insensitive` match option, but in the `with diacritics` case, only matching words in the search context nodes that contain at least one diacritical character are considered, while in the `without diacritics` case, matching words must not contain any diacritical character. Note that any diacritics in the query words have no impact on the result in both cases. For example, the following query returns books that contain the word *naive* with at least one diacritical character (such as *naïve*, *näive*, etc.):

```
//books[. ftcontains "naive" with diacritics]
```

Character wildcards—The `with wildcards` match option allows certain character sequences in a query word to be interpreted as character wildcards. The following wildcard character sequences are supported by XQFT:

- `"."` stands for any single character.
- `".?"` stands for zero or one character.
- `".*"` stands for zero or more characters.
- `".+"` stands for one or more characters.
- `".{n,m}"` stands for n to m characters (where n and m are numbers)

This notation is a subset of the regular expression notation used elsewhere in XQuery. When the `with wildcards` match option is used and a wildcard is present in a query word, it means that the query word matches a word in a document if and only if the document word can be obtained from the query word by replacing the wildcard character sequence by a sequence of arbitrary characters with a length as allowed by the wildcard. The match between query words and document words is always one to one. A wildcard, therefore, does not allow a query word to match multiple words simultaneously. If the `with wildcards` match option is not used, then wildcard characters are simply matched as regular characters in the document.

As an illustration, the following query returns books that contain at least one word that matches the query word `"eff.c.+"`, where `"."` and `".+"` are interpreted as wildcards (e.g., books that contain words such as *efficient* and *effective*):

```
//book[. ftcontains "eff.c.+" with wildcards]
```

Another interesting example is the following query:

```
//book//*[. ftcontains "site.* user."
  with wildcards]
```

It contains a multiword query term containing the wildcards `".*"` and `"."`. The tokenizer should break this up into two words (the wildcard sequences are to be considered token-internal characters). Hence, if applied to our sample document, this query returns only the note element. The word sequence *site supports the users* in the paragraph above is not matched, because `"site.*"` can only match a single word.

Thesaurus expansions—In some cases, when users issue a full-text search query with query words such as *canine*, they are also looking for results that contain semantically related words such as *dog* and *poodle*. The `with thesaurus` match option specifies such full-text search conditions, where relationships as defined in a standard thesaurus—such as synonyms, broader terms, and narrower terms (see References 34–38 for thesaurus standards)—are exploited. In XQFT, a thesaurus expansion can be specified in a query by providing three things: a Uniform Resource Identifier (URI) reference to the thesaurus to be used, the relationship to be used, and an optional depth parameter. As `with thesaurus` is a match option, it can be specified at any level of the query and applied to all query words mentioned in that part of the query. The application of a thesaurus expansion to a query word means that the full-text search is performed as though the disjunction of related words has been specified in place of the query word.

As an example, the following query returns books that contain a synonym of the word *canine*:

```
//book[. ftcontains "canine"
         with thesaurus at
         "http://bstore1.example.com/
          BSThesaurus.xml"
         relationship "synonyms"]
```

The actual technique used for thesaurus expansion is implementation defined, including whether the thesaurus URI refers to a system-defined or user-defined thesaurus.

Stop words—When performing a search, search engines often have a built-in means to disregard words that do not carry their own meaning, such as articles, prepositions, and function words (such as *but* and *if*). Such words are called *stop words*. The advantage of disregarding stop words is that queries can be processed faster (because common stop words do not need to be processed), and the returned results are of higher relevance (as stop words typically carry little meaning). Further, if stop words are not indexed, the size of the full-text indexes may be considerably smaller, depending on the kind of encoding used.[39] However, if stop words are ignored, then queries where stop words are relevant, as in the phrase query *to be or not to be*, can no longer be answered. In XQFT, the `with stop words` match option can be used to control the list of stop words to be employed. The stop words can be specified either as a URI that points to a stop word list or by directly listing the stop words in the query. In addition, the lists can be combined dynamically by using the set operations union and except. For example, the following query returns books that contain the phrase *planning then conducting* while ignoring stop words that are specified in the URI

`http://bstore1.example.com/StopWordList.xml`:

```
//book[. ftcontains "planning then conducting"
  with stop words at
  "http://bstore1.example.com/StopWordList.xml"]
```

The `with default stop words` match option can be used to select a system-provided default stop word list, and the `without stop words` match option switches off stop-word processing in the part of the query to which this option is applied.

Language option—The stemming, thesaurus, and stop words match options may not produce sensible results if the language of the documents or query words is unknown. For example, *but* in English is likely to be a stop word, whereas in French this word means *aim*, which is not likely to be a stop word. It is therefore necessary to be able to specify the language of the words in a query. In XQFT, the language is specified using the `language` match option. The following query selects the French language for language-dependent features, such as the selection of the default stop-word list:

> ■ Formally, the input and output of XQuery are defined in terms of a data model ■

```
$book[. ftcontains "salon de the"
  with default stop words language "fr"]
```

The set of valid language identifiers (such as `fr`) is implementation defined.

FTIgnore—The match options we have described so far all relate to the matching of single words or phrases. Using the `FTIgnore` option, it is possible to modify which parts of the XML structure are available for a single match of `FTSelection`. The

FTIgnore option can be specified only on the top-level FTSelection, basically extending the syntax of FTContainsExpr to the following:

```
FTContainsExpr ::= Expr "ftcontains" FTSelection
["without" "content" Expr]
```

The Expr following without content specifies a sequence of nodes, the containing text of which should be ignored when searching the search context nodes. For instance, the following query allows us to search the content element of a book, including all its descendant elements, but not including descendants of type footnote:

```
//book[. ftcontains "web site testing"
  without content .//footnote]
```

There are two aspects to this exclusion of text material from the search context. First, when the phrase *web site testing* appears in a footnote element (or descendant element thereof), it should not be found. Second, when eliminating footnotes, the distances of terms in the remaining text are affected. For instance, when ignoring a footnote that happens to stand between *web site* and *testing*, as in the example below, the terms become adjacent and can be matched as a single phrase:

```
<p>Web site<footnote>only sample.com here
  </footnote>
  testing ...</p>
```

XQFT: Scoring

The FTContainsExpr expression returns a Boolean value that indicates whether the search context nodes satisfy the full-text search condition. However, FTContainsExpr does not indicate how well the search context nodes satisfy the full-text condition. For instance, in a query that looks for books that contain the word *usability*, a book that contains one thousand occurrences of *usability* likely matches the query better than a book that contains only one occurrence of the word. In XQFT, a scoring construct is used to return a score (a floating-point value between 0 and 1, with 1 being the best match) to indicate how well search context nodes satisfy the full-text search condition. The search results can also be ranked (ordered) based on the score, so that only top-ranked results are returned to the user. This can be particularly useful when many nodes satisfy the full-text search condition.

A large number of scoring algorithms for full-text search have been proposed in the literature[20,21,30,40–42] and scoring for XML documents is still an active area of research. The goal of XQFT is not to standardize a specific scoring algorithm, but rather, to provide a language framework so that implementors can plug in the appropriate scoring algorithm for an application. Consequently, XQFT specifies only two high-level properties that every scoring mechanism must satisfy, as specified in the XQFT Requirements document:[11]

- The score values must be of type xs:float in the range [0, 1].
- For nodes in the search context that satisfy the expression used for scoring, a higher score should imply a higher degree of relevance to the expression.

Until recently, scoring has been an evolving part of XQFT, and different alternative approaches for scoring have been devised in the published W3C working drafts for XQFT over time. The approach we describe here, the extension of the XQuery FLWOR expression with scoring variables was published September 15, 2005.

This approach to scoring extends the for-clause and the let-clause in the FLWOR expression. Specifically, the for-clause and the let-clause are extended to optionally bind to a score variable, in addition to binding to a regular variable. The presence of the score variable has no effect on the value or sequence of values that get bound to the regular variable. However, parallel to the evaluation of the expression in the for-clause or let-clause that determines those values, a score value has to be computed for each such binding in an implementation-dependent way.

The following query computes a sequence of books that satisfy the FTSelection "usability" && "testing" together with scores and returns those books in descending order of their score:

```
for $book score $score in //book[. ftcontains
  "usability" && "testing"]
order by $score descending
return $book
```

Note that in this query the fact that scores are computed has no impact on which books are part of the result sequence, but only on the order in which the books are returned.

The following query returns books in decreasing order of the score of how well they match `FTSelection "usability" && "testing"`:

```
for $book in //book
let score $score := $book ftcontains
  "usability" && "testing"
order by $score descending
return $book
```

The let-clause binds `$score` to the score of `FTContainsExpr`. Here the form of the extended let-clause is used in which the regular variable that would be bound to the Boolean value of `FTContainsExpr` is omitted. Note that all books, not just the books that satisfy `FTSelection`, are returned here.

The extended let-clause enables users to filter based on one full-text search condition and score based on another full-text search condition. As an illustration, the following query returns books that contain the words *usability* and *testing* (filtering full-text search condition) ordered in descending order of their score with respect to the words *usability* and *analysis* (scoring full-text search condition):

```
for $book in
  //book[. ftcontains "usability" && "testing"]
let score $score :=
  $book ftcontains "usability" && "analysis"
order by $score descending
return $book
```

In many cases, users may not wish to obtain all the results, but only the top few highest-ranked results in terms of score. The following query returns only the top 10 highest-ranked documents with respect to the full-text search condition:

```
for $result at $rank in
for $book score $score in
  //book[. ftcontains "usability" && "testing"]
order by $score descending
return $book
where $rank <= 10
return $result
```

The inner `FLWOR` expression is the same as in the first example in this section and returns the books in descending order of their score. The outer `FLWOR` expression iterates over this ordered list of books and returns only those in the top 10 positions.

Users can also specify weights in `FTSelection` expressions, which can be used to emphasize or deemphasize different parts of the full-text search conditions in computing the score. A weight is a floating-point number and can be applied individually to each `FTSelection` expression, with higher weights meaning higher importance in computing the score. As an illustration, the following query returns books in descending order of the score, where the score is computed with a weight of 0.8 for the word *usability* and a weight of 0.3 for the word *testing*:

```
for $book score $score in
  //book[. ftcontains "usability" 0.8 &&
    "testing" 0.3]
order by $score descending
return $book
```

The exact means by which the scoring algorithm uses weights is implementation defined.

CONCLUSION AND LOOKING FORWARD

We have presented an overview of XQFT, which is being specified by the W3C for full-text search querying in XML. XQFT gracefully combines structured search (such as in XPath and XQuery) with a wide range of powerful full-text search primitives, ranging from simple term search to more complex term proximity combined with stemming and thesaurus features. The information presented in this document is intended to be in a shorter, more accessible form than the current W3C working draft.

Several open issues are still under consideration at the W3C regarding XQFT. In particular, issues related to the composability of full-text primitives are under consideration. The W3C also invites public comments on XQFT: comments can be entered into the issue tracking system (http://www.w3.org/XML/2005/04/qt-bugzilla) or, if access to that system is not possible, comments can be sent by e-mail to the W3C mailing list (E-mail: public-qt-comments@w3.org, http://lists.w3.org/Archives/Public/public-qt-comments/) with "[FT]" at the beginning of the subject field of the e-mailed message.

ACKNOWLEDGMENTS

Chavdar Botev and Jayavel Shanmugasundaram were partially supported by the Cornell/AFRL Information Assurance Institute and NSF CAREER Award IIS-0237644.

**Trademark, service mark, or registered trademark of Google Inc. or Massachusetts Institute of Technology in the United States, other countries, or both.

CITED REFERENCES

1. Initiative for the Evaluation of XML (INEX) Retrieval, http://inex.is.informatik.uni-duisburg.de/.
2. Legislative Documents in XML at the United States House of Representatives, http://xml.house.gov/.
3. The DBLP Records in XML, http://dblp.uni-trier.de/xml/.
4. Association for Computing Machinery Special Interest Group on Management of Data (ACM SIGMOD) Record: XML Version, http://www.dia.uniroma3.it/Araneus/Sigmod/.
5. The Plays of Shakespeare in XML, http://xmlcoverpages.org/bosakShakespeare200.html.
6. *XML Path Language (XPath) 2.0*, A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, Editors, W3C Working Draft (April 4, 2005), http://www.w3.org/TR/2005/WD-xpath20-20050404/.
7. *XQuery 1.0: An XML Query Language*, S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, Editors, W3C Working Draft (April 4, 2005), http://www.w3.org/TR/2005/WD-xquery-20050404/.
8. *XQuery 1.0 and XPath 2.0 Functions and Operators*, A. Malhotra, J. Melton, and N. Walsh, Editors, W3C Working Draft (April 4, 2005), http://www.w3.org/TR/2005/WD-xpath-functions-20050404/.
9. *XQuery 1.0 and XPath 2.0 Full-Text Use Cases*, S. Amer-Yahia and P. Case, Editors, W3C Working Draft (July 9, 2004), http://www.w3.org/TR/2004/WD-xmlquery-full-text-use-cases-20040709/.
10. *XQuery 1.0 and XPath 2.0 Full-Text*, S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, D. McBeath, M. Rys, and J. Shanmugasundaram, Editors, W3C Working Draft (April 4, 2005), http://www.w3.org/TR/2005/WD-xquery-full-text-20050404/.
11. *XQuery and XPath Full-Text Requirements*, S. Buxton and M. Rys, Editors, W3C Working Draft (May 2, 2003), http://www.w3.org/TR/xmlquery-full-text-requirements/.
12. P. Case, *Library of Congress Use Cases and Recommendations for XQuery Text Operators and Functions,* Presentation to the XML Government Working Group, Washington, D.C. (October 17, 2001), http://xml.gov/presentations/loc/xmlquery.htm.
13. S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TeXQuery: A Full-Text Search Extension to XQuery," *Proceedings of the 13th International World Wide Web Conference,* New York (2004), pp. 583–594.
14. GalaTex: An XML Full Text Search Engine, http://www.galaxquery.com/galatex/.
15. Cornell Database Group, The Quark Project, http://www.cs.cornell.edu/database/quark/quark_main.html.
16. D. Chamberlin, "XQuery: An XML Query Language," *IBM Systems Journal* **41**, No. 4, 597–615 (2002).
17. H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. Kay, J. Robie, M. Rys, J. Simeon, J. Tivy, and P. Wadle, *XQuery from the Experts: A Guide to the W3C XML Query Language*, H. Katz, Editor, Addison-Wesley, Boston, MA (2003).
18. *XQuery 1.0 and XPath 2.0 Data Model*, M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, Editors, W3C Working Draft (November 3, 2005), http://www.w3.org/TR/xpath-datamodel/.
19. G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York (1986).
20. N. Fuhr and K. Grossjohann, "XIRQL: An Extension of XQL for Information Retrieval," *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval,* Athens, Greece (2000), pp. 172–180.
21. A. Theobald and G. Weikum, "The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking," *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*, Prague, Czech Republic (2002), pp. 477–495.
22. E. W. Brown, "Fast Evaluation of Structured Queries for Information Retrieval," *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA (1995), pp. 30–38.
23. S. E. Robertson, "The Probability Ranking Principle in IR," *Journal of Documentation* **33**, No. 4, 294–304 (1977).
24. D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer, "Searching XML Documents via XML Fragments," *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Toronto, Canada (2003), pp. 151–158.
25. T. T. Chinenyanga and N. Kushmerick, "Expressive and Efficient Ranked Querying of XML Data," *Proceedings of the 4th International Workshop on the Web and Databases (WebDB)*, Santa Barbara, CA (2001), pp. 1–6.
26. G. Salton, A. Wong, and C. S. Yang, "A Vector Space Model for Automatic Indexing," *Communications of the ACM* **18**, No. 11, 613–620 (1975).
27. D. Florescu, D. Kossmann, and I. Manolescu, "Integrating Keyword Search into XML Query Processing," *Proceedings of the 9th International World Wide Web Conference*, Amsterdam, The Netherlands (2000), pp. 119–135.
28. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Santa Barbara, CA (2001), pp. 425–436.
29. J.-M. Bremer and M. Gertz, "XQuery/IR: Integrating XML Document and Data Retrieval," *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*, Madison, WI (2002), pp. 1–6.
30. Y. Hayashi, J. Tomita, and G. Kikui, "Searching Text-Rich XML Documents with Relevance Ranking," *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval*, Athens, Greece (2000), pp. 32–40.
31. S. Al-Khalifa, C. Yu, and H. V. Jagadish, "Querying Structured Text in an XML Database," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California (2003), pp. 4–15.
32. E. Hung, Y. Deng, and V. S. Subrahmanian, "TOSS: An Extension of TAX with Ontologies and Similarity

Queries," *Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data*, Paris, France (2004), pp. 719–730.

33. J. Melton and A. Eisenberg, "SQL Multimedia and Application Packages (SQL/MM)," *SIGMOD Record* **30**, No. 4, 97–102 (2001).
34. L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa, "A Structured Text ADT for Object-Relational Databases," *Theory and Practice of Object Systems* **4**, No. 4, 227–244 (1998).
35. The Unicode Consortium, *The Unicode Standard, Version 4.0*, Addison-Wesley Developers Press, Boston, MA (2003); for the latest version, see http://www.unicode.org/unicode/standard/versions/.
36. *ISO 5964-1985 (E), Documentation—Guidelines for the Establishment and Development of Multilingual Thesauri*, International Organization for Standardization.
37. *ISO 2788-1986(E), Documentation—Guidelines for the Establishment and Development of Monolingual Thesauri*, International Organization for Standardization.
38. *ANSI/NISO Z39.19 Monolingual Thesaurus Creation Standard*, http://www.niso.org/standards/resources/Z39-19.html.
39. I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, New York (1994).
40. W. W. Cohen, "Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA (1998), pp. 201–212.
41. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA (2003), pp. 16–27.
42. S. H. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo, "A Flexible Model for Retrieval of SGML Documents," *Proceedings of the 21st ACM SIGIR International Conference on Research and Development in Information Retrieval*, Melbourne, Australia (1998), pp. 138–145.

Accepted for publication October 28, 2005.
Published online May 17, 2006.

Sihem Amer-Yahia
AT&T Labs Research, 180 Park Avenue, Florham Park, New Jersey 07932 (sihem@research.att.com). Dr. Amer-Yahia is a Senior Technical Specialist. She received her Ph.D. degree in computer science from the University Paris XI-Orsay and INRIA, France. She has worked on various aspects related to XML query processing and recently has been focusing on XML full-text search. She is the author of various conference and journal publications and conducts tutorials on the patents discussed in this paper. She is a coeditor of the XQuery Full-Text Language Specification and Use Cases published by the W3C Full-Text Task Force. Dr. Amer-Yahia is also the leader of the GalaTex project, a conformance implementation of XQuery Full-Text.

Chavdar Botev
Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, New York 14850 (cbotev@cs.cornell.edu). Mr. Botev is a Ph.D. student in computer science. He received B.S. and M.S. degrees in informatics from Sofia University, Bulgaria. He has been an invited expert of the W3C Full-Text Task Force since 2003 and an editor of XQuery 1.0 and XPath 2.0 Full-Text since 2004.

Jochen Dörre
IBM Deutschland Entwicklung GmbH, Schönaicherstrasse 220, 71032 Böblingen, Germany (doerre@de.ibm.com). Dr. Dörre is a software engineer with a background in text search and text-mining technology. He joined IBM in 1997 and has worked on several software development projects in fields specializing on text categorization, text analytics integration, search over XML documents, and core search engine design and performance issues. He received his M.S. degree in computer science and his Ph.D. degree from the University of Stuttgart, Germany. Dr. Dörre is a member of the W3C XQuery Working Group and its Full-Text Task Force working on the XQuery and XPath Full-Text language.

Jayavel Shanmugasundaram
Department of Computer Science, Cornell University, 4105A Upson Hall, Ithaca, New York 14853 (jai@cs.cornell.edu). Dr. Shanmugasundaram is an Assistant Professor. He obtained a B.S. degree from the Regional Engineering College, Tiruchirappalli, India, an M.S. degree from the University of Massachusetts-Amherst, and a Ph.D. degree from the University of Wisconsin-Madison, all in computer science. Prior to joining Cornell University, he spent two years at the IBM Almaden Research Center. His research interests include Internet data management, information retrieval, and query processing in emerging system architectures. He is the author of several publications and patents on these topics, and his research ideas have been implemented in commercial data management products. He is an invited expert and coeditor of the XQuery and XPath Full-Text language currently being developed by the W3C. Dr. Shanmugasundaram is the recipient of a National Science Foundation CAREER Award, an IBM Faculty Award, and the James and Mary Tien Excellence in Teaching Award. ■