TeXQuery: A Full-Text Search Extension to XQuery

Sihem Amer-Yahia
AT&T Labs – Research
sihem@research.att.com

Chavdar Botev
Cornell University
cbotev@cs.cornell.edu

Jayavel Shanmugasundaram
Cornell University
jai@cs.cornell.edu

ABSTRACT

One of the key benefits of XML is its ability to represent a mix of structured and unstructured (text) data. Although current XML query languages such as XPath and XQuery can express rich queries over structured data, they can only express very rudimentary queries over text data. We thus propose TeXQuery, which is a powerful full-text search extension to XQuery. TeXQuery provides a rich set of fully composable full-text search primitives, such as Boolean connectives, phrase matching, proximity distance, stemming and thesauri. TeXQuery also enables users to seamlessly query over both structured and text data by embedding TeXQuery primitives in XQuery, and vice versa. Finally, TeXQuery supports a flexible scoring construct that can be used to score query results based on full-text predicates. TeXQuery is the precursor of the full-text language extensions to XPath 2.0 and XQuery 1.0 currently being developed by the W3C.

Categories and Subject Descriptors

H.3.m [**Information Storage and Retrieval**]: Miscellaneous

General Terms

Languages

Keywords

XQuery, full-text search

1. INTRODUCTION

One of the key benefits of XML is its ability to represent a mix of structured and unstructured (text) data. One can already find many real XML data repositories that contain such a mix of structured and text data. For example, the IEEE INEX data collection [16] contains IEEE papers in XML form, including structured information such as the names of authors, date of publication, sections, sub-sections, and references, and also unstructured information such as the text content of the paper. Other examples of such XML repositories are Shakespeare's plays in XML [4], DBLP [10] in XML, SIGMOD Record in XML [25], and the United States Library of Congress documents in XML [17]. Furthermore, application domains such as Library Science have a growing need to seamlessly query over both the structured and text parts of XML documents.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

While current XML query languages such as XPath [29] and XQuery [28] can express powerful structured queries over XML documents, they can only express a very rudimentary full-text search. For instance, full-text search in XQuery is expressed using the function: `contains($e, keywords)` which returns true iff the XML element bound to the variable `$e` contains all the keywords in `keywords` (see [33] for a precise definition of `contains`). While this function is sufficient for simple substring matching, it is woefully inadequate for more complex searches. For instance, consider the following example in the W3C XPath and XQuery Full-Text Use Cases Document [30].

Example 1: Consider an XML document that contains books. Find the titles and contents of books whose content contains the phrases "usability", "Web site" and "is" 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 used by the Information Retrieval (IR) community, such as distance predicates, synonyms, and thesauri. Finally, the `contains` function cannot score or rank results, such as returning the top 10 results for a given search.

Integrating sophisticated full-text search in XQuery introduces many challenges. First, we need to identify a set of full-text primitives that are natural to querying XML; these primitives should not only be powerful, but should also be composable with each other so that arbitrarily complex full-text searches can be specified (e.g. using stemming with distance predicates and Boolean connectives). Second, we need to leverage the full expressive power of the semi-structured nature of XML by seamlessly integrating regular XQuery with full-text search so that users can query over both structured and full-text data; this is non-trivial because structured XML queries operate on XML nodes, while by their very nature, full-text queries operate on keyword search tokens and their positions *within* XML nodes. Finally, we need to introduce the notion of ranked results in order to support threshold and top-K queries.

TeXQuery is a language extension to XQuery designed to address the above issues. TeXQuery provides a set of power-

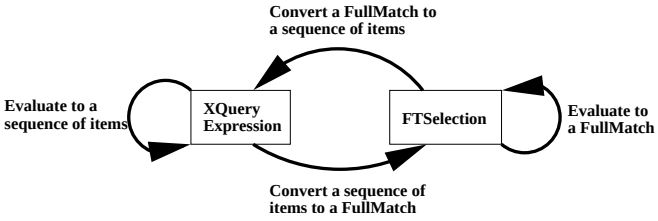

Figure 1: XQuery and TeXQuery Composability

ful full-text search primitives called `FTSelection`s. They are fully composable, and arbitrarily complex full-text queries can be created by combining the basic `FTSelection`s. The key that makes this possible (and one of the main contributions of this paper) is a formal underlying data model called *FullMatch*.

The *FullMatch* data model contains sufficient information about search tokens and their positions in an XML document such that all `FTSelection`s are closed under this data model. In other words, each `FTSelection` can be formally defined as taking in zero or more *FullMatch*es as input and produces a *FullMatch* as output. Thus `FTSelection`s can be arbitrarily composed, as shown in the right part of Figure 1. Although there have been many efforts to express full-text search on XML documents [11, 20, 1, 8, 26, 6, 18, 5, 12, 15, 26]), we are not aware of any previous data model that is closed for the same wide variety of full-text primitives.

TeXQuery can also combine full-text queries with XML queries on structure. This is achieved by two new XQuery expressions: *ftcontains* and *ftscore* (we call these the TeXQuery expressions). TeXQuery expressions specify a well-defined mapping between the *FullMatch* data model and the XQuery data model (sequence of XML items) as shown in Figure 1. Consequently, TeXQuery queries can be embedded in XQuery and vice-versa. The *ftscore* expression also enables users to score full-text search results.

TeXQuery is the precursor of the full-text language extensions to XPath 2.0 and XQuery 1.0 currently being developed by the W3C. TeXQuery satisfies all of the FTTF Requirements specified in [31], and is powerful enough to express every use case in the FTTF Use Cases document [30] (see [2] for the complete list of solutions).

The rest of the paper is organized as follows. In Section 2, we outline some design principles for XML full-text search languages. In Section 3, we describe the TeXQuery language, and in Section 4, we formally define the semantics of TeXQuery. In Section 5, we discuss related work, and in Section 6, we present some concluding thoughts.

2. DESIGN GOALS AND ALTERNATIVE APPROACHES

We now motivate and describe a set of design goals that we believe any full-text search extension to XQuery (or any XML query language in general) should satisfy. We then show why some simple extensions to the XQuery `contains` function fail to satisfy the design principles due to some fundamental limitations of the function-based approach. This motivates the need for a more powerful approach such as TeXQuery, which we describe in the next section.

We use the following terminology for the rest of this paper. A *linguistic token* is a sequence of characters that corresponds to a token in a given human language. In Western languages and many other languages, a linguistic token corresponds to a word. Leaf nodes in an XML document tree may contain multiple linguistic tokens. A *search token* is a sequence of characters defining a pattern for matching linguistic tokens. We assume that XML documents are *tokenized* by a language-dependent tokenizer to identify linguistic tokens.

2.1 Design Goals

We now describe our design goals based on the following categories.

2.1.1 Searching over Semi-Structured Data

DG1: *Users should be able to specify the search context, or the context over which the full-text search is to be performed*: In traditional full-text search [23], the search context is usually the entire document collection. However, in the case of structured or semi-structured XML documents, it is often desirable to narrow the search to a sub-set of the documents, or to fragments of documents. For instance, in the example given in the introduction, the search context is limited to books (and excludes papers, articles, etc.), and even within books, it is limited to the book content (instead of the whole book).

DG2: *Users should be able to specify the return context, or the part of the document collection that is to be returned.* In traditional full-text search [23], the return context is usually the entire document that satisfies the full-text search condition. However, in the case of structured or semi-structured XML documents, it is often desirable to return specific fragments of documents. For instance, in the example given in the introduction, the return context is limited to the title and content of books (and not other fragments of the book, such as author names, etc.).

2.1.2 Expressive power and Extensibility

DG3: *Users should be able to express complex full-text searches*. Users should be able to use sophisticated full-text primitives such as Boolean connectives, distance predicates, phrase matching, stemming, and thesauri. Further, they should be able to compose these primitives to express complex searches, such as the example in the introduction.

DG4: *The language should be extensible with respect to new full-text primitives*. Unlike the relational model, there is no general notion of "completeness" in full-text search languages. The language should thus be extensible so that new primitives (e.g. synonyms) can be added based on new user requirements.

2.1.3 Scores and Ranking

DG5: *Users should be able to obtain relevance scores for the results of full-text searches*. When searching over text, it is often desirable to rank the results based on their relevance to the search [23]. Many measures such as TF-IDF and keyword proximity can be used to obtain the relevance scores.

DG6: *Users should be able to control how scores are computed*. When issuing full-text searches, users may wish to specify that certain search tokens are more important than other search tokens [23]. For example, when searching for "XML

books", the search token XML may be more important than book, and users should be able to specify this in some way (e.g., using weights).

DG7: *Users should be able to obtain the top-K results based on their relevance score.* Since users are often interested only in the top few results, they should be able to specify this explicitly.

DG8: *Users should be able to specify a scoring condition, which is possibly different from the full-text search condition.* For example, a user may need to find all books on "software developers" and score them based on their relevance to "usability testing".

2.1.4 Integration with XQuery

DG9: *Users should be able to embed full-text searches in XQuery expressions.* This will enable users to query seamlessly over both structured data (using XQuery) and full-text data (using full-text search). This requires that full-text search expressions be fully composable with XQuery expressions.

DG10: *Users should be able to embed XQuery expressions in full-text searches.* Users should be able to use XQuery expressions to specify the search tokens for full-text search. For example, a user may wish to search for all articles that mention the title of one of Richard Dawkins' books. Here, the search tokens are the titles of Richard Dawkins' books, which are themselves the result of a XQuery query.

DG11: *XQuery's query capabilities should be leveraged wherever possible.* XQuery provides a powerful way to select, and manipulate XML documents, and this should be leveraged to avoid duplication of functionality. Some obvious ways where XQuery query capabilities can be leveraged are in the specification of the search and return contexts (DG1 and DG2).

DG12: *There should be no extensions to the XQuery data model.* Support for full-text search should have no impact on the XQuery "sequence of items" data model. The main reason is that XQuery expressions are fully compositional, and each expression takes zero or more sequences of items as input, and produces a sequence of items as output. Changing this data model (such as adding scores to items, or adding positions of search tokens) would require changing the definition of *every* XQuery expression, including those that are not full-text search expressions.

2.1.5 Language Syntax and Efficiency

DG13: *It should be possible to statically verify that a query is syntactically correct.* This is a simple requirement that states that we should be able to detect syntax errors statically (at compile time). For instance, in full-text search, we should be able to statically determine whether the Boolean operator 'and' has two operands. The main advantage, of course, is to build robust applications.

DG14: *The language syntax should allow for static type checking and inference.* Static type checking and inference are especially important when applications (not humans) interpret query results. Further, static type checking is already achieved by XQuery and it should be preserved for full-text search.

DG15: *The language should allow for an efficient implementation.* While functionality is important, language design should not preclude an efficient implementation.

2.2 Limitations of Function Approaches

We now consider two extensions to the XQuery language, which attempt to extend the basic `contains` function with more expressive full-text search capabilities. Our main goal is to illustrate that these function-based approaches have some fundamental limitations that preclude them from achieving all of the above design goals; this in turn motivates the need for a more powerful language such as TeXQuery, which we describe in the next section.

We consider two different function-based approaches. In the first approach, we create a new `contains`-like function for each full-text primitive (such as Boolean connectives, distance predicates, etc.). In the second approach, we extend the `contains` function so that this single function is used to express all full-text primitives, similar to SQL/MM [18]. Both of these approaches can be viewed as end-points in a spectrum, and there are certainly hybrid approaches that fall in between. However, the limitations of these two end-points also carry over to the hybrid approaches.

2.2.1 One Function Per Full-Text Primitive

The `contains` function checks for the occurrence of search tokens in an XML node. One can thus create other functions for other full-text operations such as Boolean connectives and distance predicates, and compose these functions to create complex full-text queries. As an example, consider the following query.

Example 2: Find all XML nodes (bound to variable $n) that contain the search token "usability" and the search token "testing". Further, the search tokens should be within a window of size 10 (i.e., a window of at most 10 tokens should contain all the search tokens).

Using a function for each Boolean connective and distance predicate, the above query can be written as:

```
distance(contains($n,'usability')
         and contains($n,'testing'), 10)
```

The function `contains($n,'usability')` returns true iff $n contains the search token 'usability', and similarly for `contains($n,'testing')`. The XQuery 'and' function is used for the Boolean connectives. Finally, a distance function operates on this result to return true only if the search tokens occur within a distance of 10.

The main problem with using this approach in the context of XQuery is that it requires an extension of the XQuery data model (thereby violating DG12). To see why this is the case, consider the return type of the first parameter of the `distance` function. The return type is Boolean because `contains` returns a Boolean value, and the Boolean connectives also return a Boolean value. But given just a Boolean value as input, how can the `distance` function determine if the search tokens are within a distance of 10 from each other? This will not be possible unless some extra information about search token positions is somehow "carried around" with the Boolean value - this is essentially a fundamental extension to the XQuery data model, violating DG12. The above problem can be avoided by disallowing distance predicates, but this would then limit the

expressive power of the language, violating DG3. It should be noted that the above problem is not pertinent exclusively to XQuery. In fact, most structured query languages for XML work at the coarse granularity of nodes.

2.2.2 Single Function for Full-Text Search

The main problem with the previous approach was that it isolated the full-text primitive into separate functions. By doing so, it had to extend the XQuery data model with position-related information so that distance-based searches can be composed. This problem can be solved by embedding the entire full-text search into a single `contains` function, such as the approach taken in SQL/MM [18]. By doing so, 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, Example 2 above can be written as follows in SQL/MM-like syntax:

```
contains($n, 'usability and testing
              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, as in the following example.

Example 3: Find all articles that mention the title of one of Richard Dawkins' books.

Here, the search tokens (the titles of Richard Dawkins' 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 DG10). One could think of generating the full-text search string "on the fly", using string concatenation on the results of XQuery expressions as follows.

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

However, this implies that the 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 above example). This violates DG13.

2.2.3 Discussion

As illustrated in the previous sections, the function-based language *syntax* has some fundamental limitations in meeting the design goals. This is unusual because, in language design, the precise syntax often does not significantly impact the expressive power or semantics. However, in our case, the syntax makes a significant difference because we are proposing an extension to an existing language (XQuery), and the syntax should fit within the framework of that language.

Of course, the syntax is just one aspect of the language. The other important aspect is its formal semantics. Even using a function-based syntax, the SQL/MM extensions do not provide the desired level of composability and semantics as outlined in our design goals (a more detailed comparison with SQL/MM can be found in Section 5). In the next two sections, we define the syntax and semantics of TeXQuery, which satisfies all of the above design goals.

3. TEXQUERY LANGUAGE

We now describe and illustrate the TeXQuery full-text search extensions to XQuery. TeXQuery satisfies all the design goals presented in Section 2.

3.1 High-Level Overview

At its core, TeXQuery introduces two new XQuery expressions, which we call TeXQuery expressions. These expressions are just like other XQuery expressions - they take zero or more sequences of items as input, and produce a sequence of items under which XQuery expressions are closed (left part of Figure 1 in the introduction). Consequently, TeXQuery seamlessly integrates with XQuery.

TeXQuery expressions support powerful full-text search by using a set of fully composable full-text primitives called `FTSelection`s. `FTSelection`s are closed under a data model that we call *FullMatch* (right part of Figure 1). The above design brings significant flexibility to TeXQuery. It can be easily restricted by removing `FTSelection`s or extended by adding new `FTSelection`s using extended `FullMatch`es. These can be achieved without modifications to the semantics of the other `FTSelection`s.

The *FullMatch* model is different from the XQuery model because full-text search, by its very nature, has to deal with linguistic tokens and their positions *within* XML nodes. We describe *FullMatch* in detail in Section 4.

It is important to note that the *FullMatch* data model is not an extension to the XQuery data model (DG12). Rather, *FullMatch* is *internal* to TeXQuery expressions. TeXQuery expressions still return a sequence of items, and are thus fully composable with other XQuery expressions (DG9 and DG10). Having a different data model *within* an XQuery expression is not specific to TeXQuery. In fact, one of the core XQuery expressions - FLWOR - has an internal model of tuples, which is not present in the XQuery data model [32].

3.2 TeXQuery Expressions

We now introduce the two TeXQuery expressions, `FTContainsExpr` and `FTScoreExpr`.

3.2.1 FTContainsExpr

The `FTContainsExpr` has the following syntax.

```
Expr ''ftcontains'' FTSelection
```

`Expr` is any XQuery expression that specifies the search context, which is the sequence of XML nodes over which the full-text search is to be performed. `FTSelection` specifies the full-text search condition. The `FTContainsExpr` returns a Boolean value that is true iff some node in the search context satisfies the full-text search condition. An example of an `FTContainsExpr` is given below.

```
//book ftcontains 'usability' && 'testing'
```

The above expression returns true iff some book in the search context `//book` (which is an XQuery expression) contains the search tokens 'usability' and 'testing'. Here `'usability' && 'testing'` is a simple example of an `FTSelection`. More complex `FTSelection`s can be specified, but we defer this discussion to a later section.

The simple example above illustrates several key points. First, it shows how `FTContainsExpr` can limit the search context, thereby satisfying DG1. Second, since `FTContainsExpr` always returns a Boolean value, it can be easily type-checked (DG14). Third, since `FTContainsExpr` returns a result in the XQuery data model (a Boolean value), it can be arbitrarily nested within other XQuery expressions thereby satisfying DG9. A concrete instantiation of this is shown in the example below.

```
//book[.//section ftcontains 'usability'
                  && 'testing']/title
```

The above query returns the titles of those books in which some section contains the search tokens 'usability' and 'testing'. Note how the `FTContainsExpr` (`.//section ftcontains 'usability' && 'testing'`) is nested within the XQuery expression `//book[ ]/title`.

There are two other points to note about the above example. First, it shows how TeXQuery can specify a return context, or the part of the selected XML items that are to be returned (DG2). In the example, the return context is only the titles of the selected books, not the contents of these books. Second, it shows how TeXQuery leverages existing XQuery constructs such as path expressions to specify the search context (`.//section`) and the return context (`/title`), thereby satisfying DG11.

3.2.2 FTScoreExpr

`FTContainsExpr` returns true iff some node in the search context satisfies the `FTSelection`. However, it does not specify how relevant the search context nodes are to the `FTSelection`. `FTScoreExpr` addresses this issue by returning a score or measure of relevance for each node in the search context (thereby satisfying DG5). `FTScoreExpr` has the following syntax.

```
Expr ''ftscore'' FTSelectionWithWeights
```

`Expr` is an XQuery expression that specifies the search context. `FTSelectionWithWeights` specifices the full-text search condition and is similar to `FTSelection`, with the added notion of weights for computing scores. `FTScoreExpr` returns a sequence of scores corresponding to each XML node in the search context sequence.

`FTScoreExpr` provides the framework for supporting different scoring mechanisms, but does not dictate the exact scoring mechanism to be used. This decision was made because it is unlikely that different implementations will agree to use the same scoring techniques. In fact, scoring for XML is an active area of research (e.g., see [9, 12, 14, 15, 19, 26]) and many vendors view their scoring technique as one of their prime differentiators. `FTScoreExpr` thus only specifies two high-level properties that every scoring mechanism should satisfy, as required in [31].

- The score of a node in the search context should be 0 iff the node does not satisfy the full-text condition specified in `FTSelectionWithWeights`. Otherwise, its score should be in the interval (0,1].
- For the nodes in the search context, a higher value of the score should imply a higher degree of relevance to `FTSelectionWithWeights`.

An example of `FTScoreExpr` is given below.

```
//book ftscore 'usability' && 'testing'
```

The above expression returns a sequence of scores for each book in the search context. The scores are computed using the `FTSelectionWithWeights` `'usability' && 'testing'`. The following example shows how the user can specify weights in the `FTSelectionWithWeights` to control how scores are computed (DG6).

```
//book ftscore 'usability' weight 0.8 &&
               'testing' weight 0.2
```

The above expression returns a sequence of scores for each book in the search context, but the score is computed using a weight of 0.8 for the search token 'usability' and a weight of 0.2 for the search token 'testing'. The exact means by which the scoring mechanism uses these weights is implementation-defined, and `FTScoreExpr` just provides the necessary language framework for specifying the weights.

Since the result of `FTScoreExpr` is a sequence of floating-point items, it can be easily type-checked (DG14). Further, since the result type is an instance of the XQuery data model, it can be arbitrarily embedded in other XQuery expressions. In particular, `FTScoreExpr` can be used in conjunction with FLWOR to compute top-K search results (DG7 and DG11). The following example illustrates how to compute the top-10 results for the previous query.

```
for $result at $rank in
 for $node in //book
 let $score := $node ftscore 'usability'
     weight 0.8 && 'testing' weight 0.2
 order by $score descending
 return <result score={$score}>
                {$node} </result>
where $rank <= 10
return {$result}
```

Finally, `FTContainsExpr` and `FTScoreExpr` can be combined to search based on one condition and score based on another condition (DG8). The following example illustrates how books can be filtered based on `'usability' && 'analysis'` and scored based on `'usability' && 'testing'`.

```
for $book in
       //book[. ftcontains 'usability'
                && 'analysis']
let $score := $book ftscore 'usability'
    weight 0.8 && 'testing' weight 0.2
return <result score={$score}>
               {$book} </result>
```

3.3 FTSelections

As mentioned above, the full-text search conditions in `FTContainsExpr` and `FTScoreExpr` are expressed in terms of an `FTSelection`. An `FTSelection` can either be a single search token (such as 'usability'), or can express more complex full-text search including Boolean connectives (and, or, not), scope of search tokens (whether they occur in the same sentence, paragraph, or node), window predicates, and number of occurrences of search tokens. In addition, `FTSelectionWithWeights` can also specify weights used

for scoring. We now illustrate some important FTSelections through examples. We specify their formal semantics in the next section. The full grammar production rules for FTSelections can be found in [2].

Consider the following FTContainsExpr.

```
//book ftcontains 'usability' &&
       'testing' same sentence window 5
```

The above expression returns true iff some book in the search context contains the search tokens 'usability' and 'testing' in the same sentence within a window of 5. Note how the simple FTSelections ('usability') and ('testing') are composed using a Boolean connective (&&) to get a more complex FTSelection ('usability' && 'testing'). This FTSelection is then composed with a scope selection (same sentence) and a window selection (window 5) to create the final FTSelection used in the above expression. This example thus illustrates how relatively complex FTSelections can be constructed by composing basic full-text primitives.

The following example illustrates another important feature of FTSelections.

```
//article ftcontains //book[./author =
          'Richard Dawkins']/title any
```

The above expression returns true if some article in the search context contains a reference to a title of one of Richard Dawkins' books. Note how an XQuery expression (//book[./author = 'Richard Dawkins']/title) is used to specify the search tokens. This shows how an XQuery expression can be embedded inside full-text search (DG10).

3.4 FTContextModifiers

FTContextModifiers can be applied on any FTSelection to modify how the full-text search is performed. FTContextModifiers specify aspects such as stemming, stop-words, regular expressions, case (upper case or lower case), diacritics, special characters, synonyms, languages, and ignoring specified XML subtrees [3]. Again, we illustrate some of the key context modifiers through examples, and refer the reader to [2] for the full details.

```
//book ftcontains 'usability' &&
       'testing' with stems
```

The above expression returns true iff some book in the search context contains the search tokens 'usability' and 'testing', using stemming (an FTContextModifier) to match the search tokens. Therefore, a book that contains 'user' and 'tests' will also satisfy the full-text search condition because both 'usability' and 'user' have the same stem ('use'), while 'testing' and 'tests' have the same stem ('test'). Note that the FTContextModifier (with stems) applies to the entire FTSelection ('usability' && 'testing') it is applied on.

A more complex example is given below.

```
//book ftcontains 'usability' &&
       'testing' with stems window 5
       without stopwords
```

The above expression returns true iff some book in the search context contains the search tokens 'usabilility' and 'testing', using stemming to match the search tokens. Further, the search tokens should appear within a window of 5, ignoring stopwords FTContextModifier when computing this window. Note how FTSelections and FTContextModifiers can be seamlessly composed.

4. TEXQUERY SEMANTICS

We now specify the formal semantics of the TeXQuery language. Our main contribution here is the *FullMatch* data model. *FullMatch* contains enough information to guarantee that full-text search primitives (FTSelections) can be closed under this model. In other words, the semantics of each FTSelection can be specified as a transformation of zero or more input *FullMatch*es to an output *FullMatch*. Therefore, *FullMatch* serves as a powerful formalism for specifying and reasoning about full-text search, similarly to the relational model that is the foundation for relational querying. Although there have been many efforts to express full-text search on XML documents [11, 20, 1, 8, 26, 6, 18, 5, 12, 15, 26]), we are not aware of any previous data model that is closed for the same wide variety of full-text primitives.

FullMatch has the following benefits. First, it ensures that FTSelections are fully composable (DG3). Second, it makes TeXQuery extensible with respect to adding new FTSelections, because each new primitive only needs to specify its semantics in terms of *FullMatch*, and does not impact the semantics of existing primitives (DG4). Third, *FullMatch* presents a clean and elegant way to specify the semantics of FTSelections. Finally, although beyond the scope of this paper, we expect that *FullMatch* will provide a principled framework for the optimization of full-text search (DG15).

FullMatch has a hierarchical structure. Thus, a *FullMatch* can be represented in XML. Consequently, the semantics of each FTSelection can be specified as a transformation from zero or more input XML *FullMatch*es into an output XML *FullMatch*. This XML-to-XML transformation can be specified in XQuery. Thus, the semantics of FTSelections can be specified in XQuery itself! Specifying the semantics of FTSelections in XQuery may enable the joint optimization of XQuery queries and full-text search.

4.1 The *FullMatch* Data Model

XQuery is based on the "sequence of items" data model [32], where an item is an XML node (or an atomic value). Since this model is defined at the granularity of XML nodes, it is inadequate for the full composability of FTSelections (see Section 2.2). We have thus developed the *FullMatch* data model based on the positions of linguistic tokens *within* XML nodes. We first introduce positions, before describing *FullMatch*.

4.1.1 Positions

A position represents the occurrence of a linguistic token in an XML document. It contains the following:

- The linguistic token
- A unique identifier that captures the relative position of the linguistic token in document order
- The XML node directly containing the linguistic token
- The relative position of the sentence containing the linguistic token
- The relative position of the paragraph containing the linguistic token

- The context of the linguistic token (e.g., tag name, attribute name, attribute value, element content)

A position can thus be modeled as an XML element conforming to the following DTD.

```
<!ELEMENT Position (Token, Identifier,
          Node, Sentence, Para, Context)>
```

The XML document in Figure 2 has been annotated to illustrate the position of each linguistic token (the positions are within parenthesis). For readability, only the unique identifier part of positions is shown.

4.1.2 FullMatch Description

A *FullMatch* is essentially a propositional logic disjunctive normal form (DNF) predicate specified using XML positions. The predicate captures the precise condition that an XML node needs to satisfy in order to be a result for a full-text search. We now illustrate *FullMatch* using examples.

Consider the `FTSelection ('usability' with stems)` evaluated over the XML document in Figure 2. The *FullMatch* corresponding to this `FTSelection` is shown in Figure 3. Here, the *FullMatch* corresponds to the entire DNF formula, each *SimpleMatch* corresponds to one of the disjuncts in the DNF formula, and each *StringInclude* corresponds to an atom in the DNF formula.

Intuitively, each *SimpleMatch* in Figure 3 represents one possible "solution" to the `FTSelection`. The "solution" described by the first *SimpleMatch* are those nodes that contain (represented as *StringInclude*) the linguistic token 'usability' in position 11. The "solution" represented by the second *SimpleMatch* are those nodes that contain the linguistic token 'users' in position 29. Note that 'users' has the same stemmed form as 'usability' (namely 'use') and is hence included in a *SimpleMatch*. Figure 4 and Figure 5 show the *FullMatch*es corresponding to the `FTSelection`s (`'software'`) and (`'Rose'`), respectively.

Note that a *FullMatch* does not directly list the nodes that satisfy an `FTSelection`. Rather, it specifies a position-based predicate that XML nodes need to satisfy in order to satisfy an `FTSelection`. By specifying a *FullMatch* in terms of positions, rather than XML nodes, there is sufficient information in a *FullMatch* to achieve full composability among `FTSelection`s. At the same time, the interpretation of a *FullMatch* as a predicate on XML nodes enables the mapping to the XQuery data model. In Figure 6, if an XML node in the search context satisfies any of the *SimpleMatches*, it qualifies as an answer.

Let us now consider a more complex example. Consider the `FTSelection ('usability' with stems && 'software')`. The corresponding *FullMatch* is shown in Figure 6. There are four possible "solutions" to this *FullMatch*, and they are represented by the four *SimpleMatch*es. The first *SimpleMatch* matches 'usability' at position 11 *and* 'software' at position 13. The second *SimpleMatch* matches 'usability' at position 11 and 'software' at position 18, etc.

As a final example, consider the `FTSelection ('usability' with stems && 'software' && !'Rose')`. Here "!" is the Boolean 'not' operator used to specify the absence of a search token (in this case `'Rose'`). The corresponding *FullMatch* is shown in Figure 7. As in the previous example, there are four possible "solutions" (*SimpleMatch*es). However, besides *StringInclude*s, each *SimpleMatch* also has a *StringExclude* corresponding to the negated search token. A *StringExclude* specifies a position that *should not* occur in an XML node for it to be a result; this corresponds to a negated atom in the DNF formula.

4.1.3 Representing FullMatch in XML

Since *FullMatch* has a hierarchical structure, it can be represented as XML. As mentioned earlier, this allows us to specify the semantics of `FTSelection`s using XQuery itself. The DTD of the XML representation of a *FullMatch* is given below.

```
<!ELEMENT FullMatch (SimpleMatch)*>
<!ELEMENT SimpleMatch (StringInclude|
                       StringExclude)*>
<!ELEMENT StringInclude Position>
<!ELEMENT StringExclude Position>
```

4.2 Semantics of TeXQuery Expressions

We now specify the formal semantics of `FTContainsExpr` and `FTScoreExpr`. In specifying the semantics, we make use of the following two implementation-defined functions.

```
function fts:containsPos($node as node,
           $position as fts:Position)
         as xs:Boolean
function fts:score($node as node,
           $ftselection as
           fts:FTSelectionWithWeights)
         as xs:double
```

The function `fts:containsPos` returns true iff the node `$node` contains the position `$position`. The function `fts:score` returns a floating point score in the interval (0,1] for the node `$node` with respect to the `FTSelectionWithWeights` (`$ftselection`). These implementation-defined functions are designed to provide flexibility to a TeXQuery implementation, while still ensuring precise semantics.

4.2.1 Semantics of FTContainsExpr

As described in Section 3.2.1, a `FTContainsExpr` specifies a search context and an `FTSelection`, and returns true iff some node in the search context satisfies the `FTSelection`. Since the search context is an XQuery expression, it returns a sequence of XML nodes. The `FTSelection` returns a *FullMatch*. We now specify the semantics of `FTContainsExpr`, which provides the "glue" between the sequence of items and the *FullMatch* to produce a Boolean result. Since the *FullMatch* can be represented as XML, we use an XQuery function to specify this transformation.

```
function FTContainsExpr(
           $searchContext as node*,
           $fullMatch as fts:FullMatch)
         as xs:Boolean {
   some $node in $searchContext
   satisfies some $simpleMatch in
              $fullMatch/simpleMatch
     satisfies every
       $stringInclude in
       $simpleMatch/stringInclude
         satisfies fts:containsPos(
```

```
<book(1) id(2)=‘‘1000(3)’’>
  <author (4)>Elina(5) Rose(6)</author(7)>
  <content(8)>
    <p(9)> The(10) usability(11) of(12) software(13) measures(14) how(15)
           well(16) the(17) software(18) provides(19) support(20) for(21)
           quickly(22) achieving(23) specified(24) goals(25). </p(26)>
    <p(27)>The(28) users(29) must(30) not(31) only(32) be(33) well-served(34),
           but(35) must(36) feel(37) well-served(38).</p(39)>
  </content(40)>
</book(41)>
```

Figure 2: Positions Example

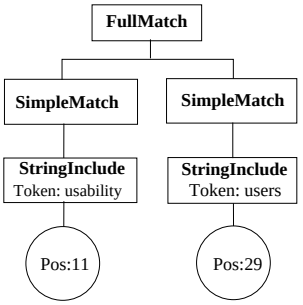

Figure 3: *FullMatch* for `'usability' with stems`

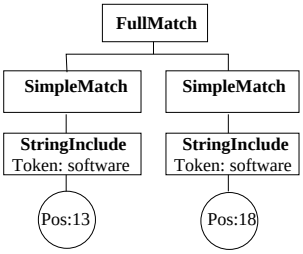

Figure 4: *FullMatch* for `'software'`

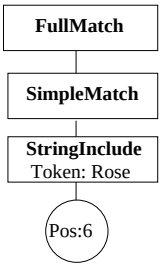

Figure 5: *FullMatch* for `'Rose'`

```
                $node,
                $stringInclude/position)
        and
        every $stringExclude in
              $simpleMatch/stringExclude
        satisfies not fts:containsPos(
                $node,
                $stringExclude/position)
}
```

The above function returns true iff some node in the search context satisfies at least one of the *SimpleMatch*es. A node is said to satisfy a *SimpleMatch* iff it satisfies all of the *StringInclude*s, and satisfies none of the *StringExclude*s.

In the example in Figure 2, the `FTContainsExpr` (`//book ftcontains 'usability' with stems && 'software'`) will return true because the `book` node satisfies at least one of the *SimpleMatch*es in Figure 6 (in fact, it satisfies all of the *SimpleMatch*es in this particular example). However, the `FTContainsExpr` (`//book ftcontains 'usability' with stems && 'software' && !'Rose'`) will return false because the `book` node does not satisfy any of the *SimpleMatch*es in Figure 7 (due to the presence of the *StringExclude*s).

4.2.2 Semantics of FTScoreExpr

As described in Section 3.2.2, an `FTScoreExpr` returns a score for every node in the search context, which is computed based on an `FTSelectionWithWeights`. Its semantics is specified below.

```
function FTScoreExpr(
          $searchContext as node*,
          $fullMatch as fts:FullMatch,
          $ftselection as
              fts:FTSelectionWithWeights)
        as xs:float {
  for $node in $searchContext
  return if FTContainsExpr($node,
                           $fullMatch)
         then fts:score($node,
                        $ftselection)
         else 0
}
```

The function returns a score of 0 for a node in the search context if the node does not satisfy the `FTSelectionWithWeights`. Else it returns a score in the interval (0,1] using a call to the implementation-defined function `fts:score`.

4.3 Semantics of `FTSelection`s

In specifying the semantics of `FTSelection`s, we use the following implementation-defined functions.

```
function fts:getPositions(
           $searchContext as node*,
           $searchToken as xs:string)
         as fts:Position*
function fts:posDistance(
           $pos1 as Position,
           $pos2 as Position,
           $ignorepos as Position*)
         as xs:integer
```

The function `fts:getPositions` returns the positions in which a search token appears in the search context; this is usually implemented using inverted lists [23]. The function `fts:posDistance` returns the distance between two positions; this distance is the number of other search tokens that occur between the two positions plus one. In computing this distance, some intervening token positions are ignored if they appear in `$ignorepos`.

We now specify the semantics of some key `FTSelection`s. The details of the other `FTSelection`s can be found in [2]. It is important to note that these definitions in terms of *FullMatch* DNF formulae is primarily for expressing the precise *semantics* of `FTSelection`s. An *implementation* can be (and probably should be) more efficient so long as it preserves this semantics.

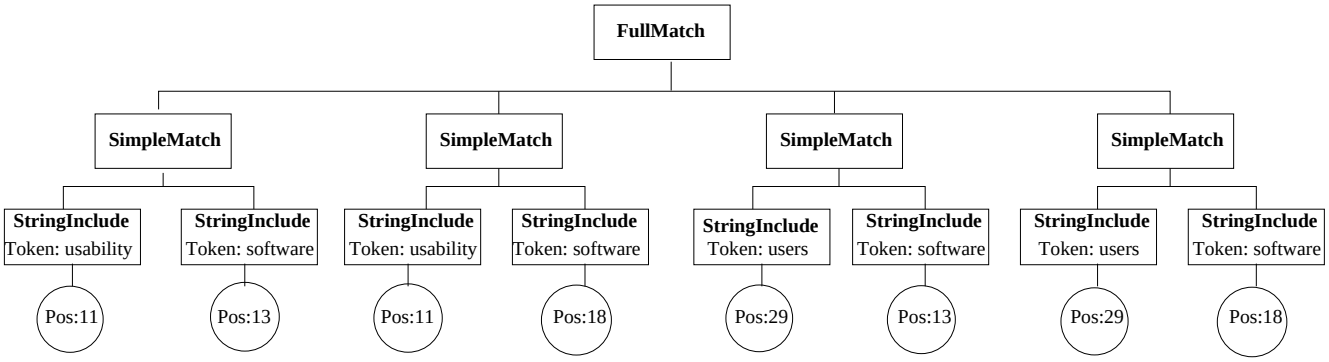

Figure 6: *FullMatch* for `'usability' with stems && 'software'`

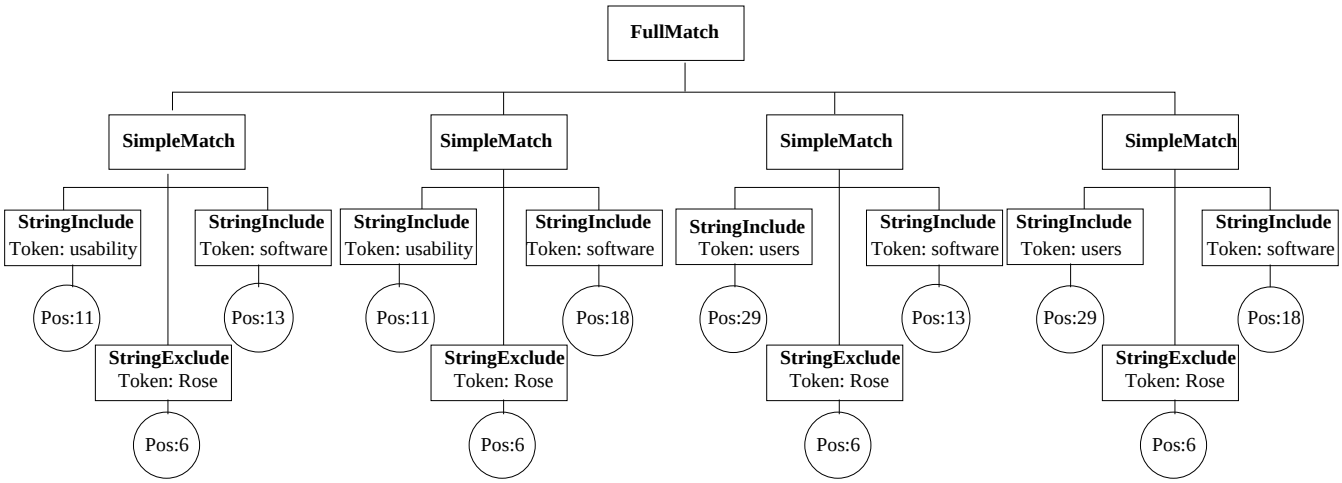

Figure 7: *FullMatch* for `'usability' with stems && 'software' && !'Rose'`

4.3.1 Semantics of `FTStringSelection`

`FTStringSelection` is the basic `FTSelection` that specifies search tokens. Its syntax is:

```
FTStringSelection ::= Expr
```

`Expr` is an XQuery expression that returns a sequence of string items. These items are used as the search tokens in the `FTStringSelection`. For ease of exposition, we limit ourselves to the case where `Expr` is a string literal that corresponds to a single search token (other cases are discussed in [2]). The semantics of how `FTStringSelection` transforms a search token into a *FullMatch* is specified by the following XQuery function.

```
function fts:FTStringSelection(
            $searchContext as node*,
            $searchToken as xs:string,
            $contextModifiers as
                fts:ContextModifier*)
         as fts:FullMatch {
  <fullMatch>
     {for $newSearchToken in
                fts:expandSearchToken(
                   $searchToken,
                   $contextModifiers),
          $position in
                fts:getPositions(
                   $searchContext,
                   $newSearchToken)
      return <simpleMatch>
                <stringInclude>
                    {$position}
                </stringInclude>
             </simpleMatch>}
  </fullMatch>
}
```

First, the `fts:expandSearchToken` function (defined precisely in [2]) takes in the given search token and the relevant context modifiers, and produces an expanded set of search token based on the context modifiers. For example, consider the `FTSelection` `'usability' with stems`. The context modifier (`with stems`) applies to the `FTStringSelection` (`'usability'`). Therefore, the search token `'usability'` is expanded to include all search tokens that have the same stem as `'usability'` (including `'usability'`, `'users'`, `'useful'`, etc.).

Given the new (expanded) set of search tokens, the position of each of these search tokens in the search context is determined using the `getPositions` implementation-defined function. Finally, a *SimpleMatch* is created for each such po-

sition, and these are nested under the result *FullMatch*.

As an illustration, the `FTSelection ('usability' with stems)` produces the *FullMatch* shown in Figure 3. The `FTStringSelections ('software')` and `('Rose')` produce the *FullMatch*es in Figure 4 and Figure 5, respectively.

Besides the stemming context modifier (discussed above), the `fts:expandSearchToken` function is also defined for other modifiers such as regular expressions, case, diacritics, special characters, and thesauri (see [2]). It is important to note that the notion of expanding search tokens is only used for specifying the *semantics* of an `FTStringSelection`. An actual *implementation* may not actually expand search tokens, so long as it produces the same results as the formal semantics. For example, stemming may be implemented by building inverted lists on stemmed forms of search tokens.

4.3.2 *Semantics of* `FTNegation`

`FTNegation` is an `FTSelection` that is used to specify Boolean negation. It can be applied on any `FTSelection` and has the following syntax.

```
FTNegation ::= ''!'' FTSelection
```

The semantics of `FTNegation` can be specified as a transformation of the *FullMatch* associated with the input `FTSelection` into the output *FullMatch*. This transformation is performed by negating the DNF formula of the input *FullMatch*, and producing the resulting output *FullMatch*. This transformation can be expressed naturally in XQuery, but since this specification is straightforward but tedious and not particularly illustrative in the current context, it is omitted here (see [2] for details). Instead, we illustrate the main idea using an example.

Consider the `FTNegation !'Rose'`. The *FullMatch* corresponding to the `FTStringSelection 'Rose'` (Figure 5) is negated to produce the resulting *FullMatch* in Figure 8. Note how *StringInclude*s become *StringExclude*s (and vice versa); this corresponds to the negation of atoms in the DNF formula corresponding to a *FullMatch*.

4.3.3 *Semantics of* `FTAndConnective`

The `FTAndConnective` combines two `FTSelection`s with the semantics of a Boolean 'and'. It has the following syntax.

```
FTAndConnective ::=
        FTSelection ''&&'' FTSelection
```

The following function specifies the semantics of `FTAndConnective` in terms of how it transforms the two input *FullMatch*es into the output *FullMatch*.

```
function fts:FTAndConnective(
           $fm1 as fts:FullMatch,
           $fm2 as fts:FullMatch)
         as fts:FullMatch {
 <fullMatch>
    {for
      $simpleMatch1 in $fm1/simpleMatch,
      $simpleMatch2 in $fm2/simpleMatch
     return <simpleMatch>
              {$simpleMatch1/*
               $simpleMatch2/* }
            <simpleMatch>}
 </fullMatch>
}
```

Each *SimpleMatch* in the resulting *FullMatch* is a combination of one *SimpleMatch* from the first input *FullMatch* and one *SimpleMatch* from the second input *FullMatch*. The intuition is that each input *FullMatch* is satisfied iff at least one of its *SimpleMatch*es is satisfied. Therefore, an 'and' of the input *FullMatch*es is satisfied iff at least one of the *SimpleMatch*es from the first input *and* one of the *SimpleMatch*es from the second input is satisfied.

The *FullMatch* for the `FTAndConnective ('usability' with stems && 'software')` is shown in Figure 6. This *FullMatch* is obtained by combining the *FullMatch*es for `'usability' with stems` (Figure 3) and for `'software'` (Figure 4). Similarly, the *FullMatch* in Figure 7 is obtained by combining the *FullMatch*es in Figures 6 and 8.

4.3.4 *Semantics of* `FTScopeSelection`

`FTScopeSelection` limits the scope of an `FTSelection` to a node, sentence, or paragraph. It has the following syntax.

```
FTScopeSelection ::= FTSelection(
                      ''same''|
                      ''different'')
                     (''node''|
                      ''sentence''|
                      ''para'')
```

The `FTScopeSelection` takes the *FullMatch* corresponding to its input `FTSelection`, and restricts the `SimpleMatch`es so that only those that have positions in the same (or different) node, sentence or paragraph are selected for the output *FullMatch*. The semantics for the `FTScopeSelection ('same para')` is given below.

```
function fts:FTParaScopeSelection(
           $fullMatch as fts:FullMatch)
         as fts:FullMatch {
 <fullMatch>
   {for $simpleMatch in
        $fullMatch/simpleMatch
    where
     every $strInclude1 in $simpleMatch,
           $strInclude2 in $simpleMatch
     satisfies
       $strInclude1/position/para =
       $strInclude2/position/para
    return
     <simpleMatch>
      {$simpleMatch/stringInclude}
      {for $stringExclude in
           $simpleMatch/stringExclude
       where every
         $stringInclude in
         $simpleMatch/stringInclude
         satisfies
          $stringInclude/position/para =
          $stringExclude/position/para
       return $stringExclude}
     </simpleMatch>}
 </fullMatch>
}
```

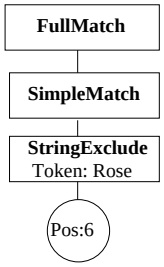

Figure 8: *FullMatch* for `!'Rose'`

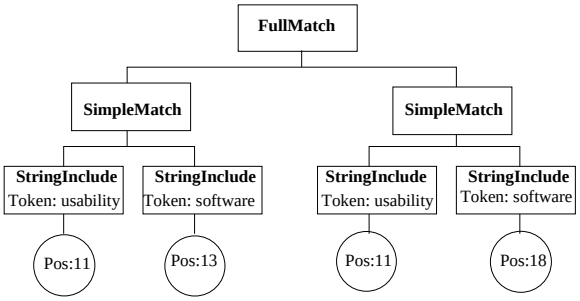

Figure 9: *FullMatch* for `'usability' with stems && 'software' && !'Rose' same para`

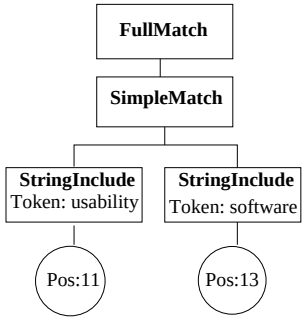

Figure 10: *FullMatch* for `'usability' with stems && 'software' && !'Rose' same para window 5 without stopwords`

As shown above, only the *SimpleMatch*es in which all the *StringInclude*s are in the same paragraph are selected for the output *FullMatch*. Further, the *StringExclude*s in the selected *SimpleMatch*es are also restricted to be in the same paragraph as the *StringInclude*s in the output *FullMatch*.

Figure 9 shows the *FullMatch* for the `FTScopeSelection ('usability' with stems && 'software' same para)`. This *FullMatch* is obtained by transforming the input *FullMatch* corresponding to `'usability' with stems && 'software'` (Figure 6). Note how the *StringExclude*s do not appear in the result *FullMatch* because they do not appear in the same paragraph as the *StringInclude*s.

4.3.5 *Semantics of* `FTWindowSelection`

`FTWindowSelection` specifies the maximum window size for an `FTSelection`. Its syntax is:

```
FTWindowSelection ::=
    FTSelection ''window'' xs:integer
```

The `FTWindowSelection` takes the *FullMatch* corresponding to its input `FTSelection`, and restricts the *SimpleMatch*es so that only those that fit in the specified window size are selected for the output *FullMatch*. This semantics is specified below.

```
function fts:FTWindowSelection(
           $fullMatch as fts:FullMatch,
           $windowSize as xs:integer,
           $contextModifiers
              as fts:ContextModifier*)
        as fts:FullMatch {
 <fullMatch>
  {let $ignorePos :=
         fts:getIgnorePos(
           $contextModifiers)
   for $simpleMatch
       in $fullMatch/simpleMatch
   where
    every $strInclude1 in $simpleMatch,
          $strInclude2 in $simpleMatch
         satisfies
            fts:posDistance(
              $strInclude1/position,
              $strInclude2/position,
              $ignorePos) < $windowSize
   return
    <simpleMatch>
      {$simpleMatch/stringInclude}
      {for $stringExclude in
           $simpleMatch/stringExclude
       where every
             $stringInclude in
             $simpleMatch/stringInclude
             satisfies
             fts:posDistance(
               $stringInclude/position,
               $stringExclude/position,
               $ignorePos) < windowSize
       return $stringExclude}
    </simpleMatch>}
 </fullMatch>
}
```

As shown above, only the *SimpleMatch*es in which all the *StringInclude*s occur within the specified window size are selected. Further, the *StringExclude*s in the selected *SimpleMatch*es are also restricted to occur within the specified window size in the output *FullMatch*. Certain search token positions ($ignorePos) are ignored when computing the distance between two positions in a *SimpleMatch*. The positions to be ignored depend on the stop-word and ignore XML subtree context modifiers; this is computed using the `fts:getIgnorePos` function (details are in [2]).

Figure 10 shows the *FullMatch* for the `FTScopeSelection('usability' with stems && 'software' && !'Rose' same para window 5 without stopwords)`. This *FullMatch* is obtained by transforming the *FullMatch* for `'usability' with stems && 'software' same para` (Figure 9), and ignoring the positions of stopwords when computing the window size.

5. RELATED WORK

The topic of combining full-text search with structured querying has recently received a lot of attention, both in research and in the industry. In research, many efforts have focused on extending XML query languages with full-text search. However, unlike TeXQuery, previous solutions explore only a few full-text search primitives at a time (e.g., Boolean keyword

retrieval [11, 20, 1], keyword similarity [8, 26], proximity distance [6, 18], relevance ranking [5, 12, 15, 26]). Further, previous techniques do not develop a fully compositional model for full-text search (such as *FullMatch*), and also do not provide a seamless integration with XQuery.

In the industry, the W3C Full-Text Task force (FTTF) has been specifically created to enhance XQuery and XPath with full-text search [30, 31]. SQL/MM [18] was designed to extend SQL to express queries on text, images and spatial data (see also [7] for a related ADT-based approach). Full-text queries are expressed in a sub-language embedded in a function call. As discussed in Section 2, the function call approach has some fundamental limitations when used in XQuery. Further, SQL/MM does not provide a fully compositional full-text data model, and does not consider integration with XQuery.

Various models have been proposed in the IR literature, including the Vector space model [24] and probabilistic models [21, 27]. These models provide a systematic way to compute the relevance of a document to a query. While TeXQuery does not dictate the use of a particular relevance model, it is flexible enough to accommodate the different models in the context of the *ftscore* expression. In sum, TeXQuery primitives span the space between pure Boolean search and complex relevance search thereby providing expressive IR search over XML documents.

6. CONCLUSION

We have presented TeXQuery, a full-text search language extension to XQuery. TeXQuery supports a powerful set of fully composable full-text search primitives, which can be seamlessly integrated into the XQuery language. We have also developed the *FullMatch* data model for formally reasoning about full-text searches. Using *FullMatch* we have formally specified the semantics of TeXQuery in terms of XQuery itself. TeXQuery is the precursor of the full-text language extensions to XPath 2.0 and XQuery 1.0 currently being developed by the W3C.

In this paper, we have focused on the TeXQuery language design and underlying formal model. We are currently developing a reference implementation of TeXQuery in Galax [13]. We are also exploring efficient query optimization and evaluation techniques based on the interactions between the XQuery and *FullMatch* data models. We are exploring methods to integrate existing ranking schemes [5, 12, 15, 26] in our model.

7. ACKNOWLEDGEMENTS

Jonathan Robie gave valuable suggestions and feedback regarding scoring and other aspects of TeXQuery. Don Chamberlin and Mary Fernández provided detailed and insightful comments on an earlier draft of this paper.

Chavdar Botev and Jayavel Shanmugasundaram were partially supported by NSF CAREER Award IIS-0237644 and an IBM Faculty Award.

8. REFERENCES

[1] V. Aguilera et al. Xyleme Query Architecture. WWW 2001.

[2] S. Amer-Yahia, C. Botev, J. Robie and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. http:/www.cs.cornell.edu/database/TeXQuery/.

[3] S. Amer-Yahia, et al. Phrase Matching in XML. VLDB 2003.

[4] J. Bosak. The plays of Shakespeare in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html.

[5] J. M. Bremer, M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. WebDB 2002.

[6] E. W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. SIGIR 1995.

[7] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, F. W. Tompa. A Structured Text ADT for Object-Relational Databases. Theory and Practice of Object Systems 4(4), 1998.

[8] T. T. Chinenyanga, N. Kushmerick. Expressive and Effi cient Ranked Querying of XML Data. SIGIR Workshop on XML and Information Retrieval, 2001.

[9] W.W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. SIGMOD 1998.

[10] DBLP in XML. http://dblp.uni-trier.de/xml/.

[11] D. Florescu, D. Kossmann, I. Manolescu. Integrating Keyword Search into XML Query Processing. WWW 2000.

[12] N. Fuhr, K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. SIGIR Workshop on XML and Information Retrieval, 2000.

[13] Galax. http://db.bell-labs.com/galax/.

[14] L. Guo et al. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[15] Y. Hayashi, J. Tomita, G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. SIGIR Workshop on XML and Information Retrieval, 2000.

[16] Initiative for the Evaluation of XML Retrieval. http://www.is.informatik.uni-duisburg.de/projects/inex03/.

[17] The Library of Congress. http://lcweb.loc.gov/crsinfo/xml/.

[18] J. Melton, A. Eisenberg. SQL Multimedia and Application Packages (SQL/MM). SIGMOD Record 30(4), 2001.

[19] S.-H. Myaeng, D.-H. Jang, M.-S. Kim, Z.-C. Zhoo. A Flexible Model for Retrieval of SGML Documents. SIGIR 1998.

[20] J. Naughton, et al. The Niagara Internet Query System. IEEE Data Engineering Bulletin 24(2), 2001.

[21] S. Robertson. The probability ranking principle in IR. Journal of Documentation 33, 1977.

[22] M. Rys. Full-Text Search with XQuery: A Status Report. In Intelligent Search on XML, Springer-Verlag, 2003.

[23] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[24] G. Salton and A. Wong. A Vector Space Model for Automatic Indexing. Communications of the ACM 18, 1975.

[25] Sigmod Record in XML. http://www.acm.org/sigmod/record/xml/.

[26] A. Theobald et al. Adding Relevance to XML. WebDB 2000.

[27] H. Turtle, B. Croft. Inference Networks for Document Retrieval. SIGIR 1990.

[28] The World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3.org/TR/xquery/.

[29] The World Wide Web Consortium. XML Path Language (XPath) 2.0. W3C Working Draft. http://www.w3.org/TR/xpath20/.

[30] The World Wide Web Consortium. XQuery and XPath Full-Text Use Cases. W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-use-cases/.

[31] The World Wide Web Consortium. XQuery and XPath Full-Text Requirements. W3C Working Draft. http://www.w3.org/TR/xmlquery-full-text-requirements/.

[32] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft. http://www.w3.org/TR/xpath-datamodel/.

[33] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft. http://www.w3.org/TR/xquery-operators/.