GalaTex: A Conformant Implementation of the XQuery Full-Text Language

Emiran Curtmola [†] Sihem Amer-Yahia [§] Philip Brown [§] Mary Fernández [§]

[†] University of California San Diego
9500 Gilman Drive
La Jolla, CA 92093
ecurtmola@cs.ucsd.edu

[§] AT&T Labs Research
180 Park Ave
Florham Park, NJ 07932
{sihem,pebrown,mff}@research.att.com

ABSTRACT

We describe GALATEX [10], the first complete implementation of XQuery Full-Text, a W3C specification that extends XPath 2.0 and XQuery 1.0 with full-text search capabilities. XQuery Full-Text provides composable full-text search primitives such as simple keyword search, Boolean queries, and keyword-distance predicates. GALATEX is intended to serve as a reference implementation for XQuery Full-Text and as a platform for addressing new research problems such as scoring full-text query results, optimizing XML queries over both structure and text, and evaluating top-k queries on scored results. GALATEX is an all-XQuery implementation initially focused on completeness and conformance rather than on efficiency. We describe its implementation on top of Galax, a complete XQuery implementation and identify some performance challenges, possible solutions, and their interactions with XQuery implementations.

1. INTRODUCTION

The ability to search both the structure and text content of XML documents is gaining importance with the increase of large XML repositories such as the United States Library of Congress documents [20], medical data in XML such as HL7 [16], and the IEEE INEX data collection [19]. Querying XML repositories rich in text content requires sophisticated full-text search features ranging from matching individual keywords to combining matches with Boolean operators and with word distances, stemming, and stop words.

XML querying is a well-studied topic, with several powerful database-style query languages such as XPath 2.0 [29] and XQuery 1.0 [28] set to become W3C standards. The XQuery expression given below is a typical query on the US Library of Congress repository that selects congressional bills with actions that relate to "non-immigrant status", such as bills that amend the Immigration and Nationality Act. The query returns the descriptions of such bills that have been introduced since 2002:

```
for $b in //bill
where fn:contains($b//action, "non-immigrant status")
      and $b//action-year >= 2002
return <bill> {$b/description} </bill>
```


Informal Proceedings of the *Second International Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, June 16-17, 2005, Baltimore, Maryland USA.

The query applies the XQuery substring-matching function *fn:contains* [34] to the text nodes contained in `action` elements. As discussed in [1], sub-string functions in XQuery cannot express more complex full-text queries, such as restricting the order and distance between words. These limitations are due to the XQuery data model, which does not represent positions of words in input documents. Word positions are necessary to compute distance and to evaluate order predicates. Therefore, even if custom XQuery functions were defined for each full-text search primitive, they would not be fully composable without extending the data model.

XQuery Full-Text [30] is an extension of XQuery that supports fully composable full-text search primitives defined on a data model of words and positions. The language is inspired by TeXQuery [1], a proposal to the W3C Full-Text Task Force. XQuery Full-Text provides powerful full-text search primitives such as simple word search, Boolean queries, word distance as well as stemming, regular expressions and stop words. XQuery Full-Text also supports scoring and top-k ranking of query results. We refer to the XQuery Full-Text search primitives as *FTSelections*. All *FTSelections* are defined on a data model, called *AllMatches*, which represents words and their positions in documents. Because the semantics of each *FTSelection* is defined in terms of operators on the *AllMatches* data model, the *FTSelections* are fully composable.

The key problems when implementing XQuery Full-Text are : (i) choosing a representation for the *AllMatches* data model; (ii) implementing the semantics of each full-text primitive on *AllMatches*; and (iii) processing input documents to provide the word positions used in *AllMatches*. In GALATEX,[1] our strategy is to employ XML and XQuery directly to solve these problems. First, we implement the *AllMatches* data model in XML itself [30]. Second, we implement each full-text primitive as a native XQuery function that takes one or more *AllMatches* values and produces an *AllMatches* value. Last, we pre-process each input document to produce auxiliary XML documents that map each word to their positions in the input documents; these auxiliary documents are accessed by the semantic functions. This implementation strategy is both general and expedient. By using XML and XQuery themselves to implement XQuery Full-Text, we were able to rapidly prototype a complete implementation of the language. In addition, our technique can be used with any XQuery implementation.[2]

[1] *http://www.galaxquery.org/galatex*

[2] See *http://www.w3.org/XML/Query* for a list of XQuery implementations.

XQuery Full-Text supports scoring and ranking of query results and permits any ranking method that satisfies the XQuery Full-Text scoring requirements [30, 32]. In GALATEX, we adapt the probabilistic relational algebra [14, 23] to *AllMatches* by extending each full-text primitive with the ability to manipulate scores. Our implementation satisfies the XQuery Full-Text scoring requirements.

When implementing GALATEX, we have focused more on completeness and conformance than on efficiency. By focusing on completeness, GALATEX can serve as a reference implementation of XQuery Full-Text and as a platform for experimenting with new research ideas for scoring XML data, optimizing XML queries on both structure and content, and evaluating top-k queries. Ultimately, we want GALATEX to be both complete *and* efficient. One of GALATEX's performance bottlenecks is the size of the *AllMatches* values generated by each *FTSelection*. We discuss several ways of optimizing the evaluation of *FTSelection*s, including logical rewritings of the full-text query and the optimization of XML queries on both structure and content.

In particular, this paper makes the following contributions:

- We present a general technique for implementing XQuery Full-Text using an existing XQuery implementation.
- We describe GALATEX, the first complete implementation of XQuery Full-Text. GALATEX is implemented almost entirely in XQuery itself. In addition to a command-line interface, GALATEX includes a browser interface that permits users to execute both the XQuery Full-Text use cases [31] and their own queries.
- We adapt the scoring method for the probabilistic relational algebra [14, 23] to *AllMatches* and show that this adaptation satisfies XQuery Full-Text's scoring requirements.
- We identify some performance challenges, possible solutions, and their interactions with existing XQuery implementations.

We begin with an overview of XQuery Full-Text in Section 2. Section 3 describes general implementation techniques and their realization in GALATEX. Advanced evaluation strategies are considered in Section 4. We conclude and present the related work in Section 5.

2. THE XQUERY FULL-TEXT LANGUAGE

We introduce XQuery Full-Text search and scoring through examples and highlight some key features of the language. We refer the reader to the language specification [30] and the language use cases [31] for more details on the language.

2.1 Full-Text Search

XQuery Full-Text extends XQuery with a full-text search expression (*FTContainsExpr*) and with a scoring function (*ft:score()*). The *FTContainsExpr* takes an evaluation context (i.e., a sequence of XML nodes) and a full-text search (*FTSelection*) condition and returns a Boolean value that is true if and only if some node in the evaluation context satisfies the condition. Because *FTContainsExpr* is a first-class XQuery expression, full-text search is seamlessly integrated into XQuery and XPath. In particular, since *FTContainsExpr* returns a value in the XQuery data model (i.e., a Boolean value), it can occur wherever a Boolean value is permitted in other XQuery expressions. The following expression illustrates the interaction of full-text search with an XPath expression.

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

The expression returns the titles of books with at least one section that contains the search tokens *usability* and *testing*. The *FTContainsExpr* is used as a predicate that returns a Boolean value. Its evaluation context is an XQuery expression, i.e., *.//section* within *//book*, and its *FTSelection* is *"usability" && "testing"*. This query also illustrates how XQuery Full-Text uses existing XQuery constructs such as path expressions to specify the evaluation context and the returned nodes (*/title*).

An *FTSelection* may be used to express matching individual words (*FTWord*), Boolean connectives between keywords (*FTAnd*, *FTOr* and *FTNegation*), order predicates (*FTOrdered*), proximity distance between words (*FTDistance* and *FTWindow*), scoping within sentences and paragraphs (*FTScope*) and the ability to specify the number of occurrences of words (*FTTimes*). The query below illustrates how these primitives can be combined. It returns true if some book in the evaluation context (*//book*) contains the tokens *usability* and *testing* in the same sentence within a window of five words.

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

XQuery Full-Text can also embed XQuery expressions. The expression below returns true if some article in the evaluation context contains an occurrence of a title of one of Paul Auster's books. The XQuery expression *//book[./author = "Paul Auster"]/title* specifies the search tokens, and the keyword *any* specifies that at least one of the titles can occur in the articles.

```
//article ftcontains
  (//book[./author = "Paul Auster"]/title) any
```

In addition to *FTSelection*s, XQuery Full-Text has a rich set of matching modifiers called, *FTMatchOptions*, such as stemming, stop-words, regular expressions, case sensitivity, diacritics, special characters, synonyms, languages, and ignoring specified XML subtrees [3]. *FTMatchOptions* operate at the level of individual words and can be seamlessly composed with any *FTSelection* to modify how the full-text search is performed. The expression below returns true if some book in the evaluation context contains any tokens derived from *usability* and *testing* after applying stemming. For example, a book that contains *user* and *tests* would satisfy the full-text search condition because *usability* and *user* share the stem *use*, and *testing* and *tests* share the stem *test*.

```
//book ftcontains
  "usability" && "testing" with stemming
```

The last example below is similar to the one above, but requires that search tokens occur within a window of five words, ignoring stop-words when computing this window.

```
//book ftcontains
  "usability" && "testing"
    with stemming window 5 without stopwords
```

2.2 Full-Text Scoring

The previous expressions all yield Boolean values, but often users require the results of full-text search to be scored and ranked by the quality of the match. In XQuery Full-Text, scoring is achieved using the second-order function *ft:score()*, which returns one score for each node in the set of input XML nodes. This function is second order because it accepts an *FTSelection* expression, not a value, as an argument – it is also the only second-order function in XQuery.

The score of a node captures its relevance to an *FTSelection*. For example, the expression below returns a sequence of scores for each book in the evaluation context.

```
let $scores := ft:score(//book,
    "usability" weight 0.8 && "testing" weight 0.2)
```

Note that user-specified weights can be applied to compute score. In this example, *usability* is given a weight of 0.8 and *testing*, a weight of 0.2. The exact means by which *ft:score* uses these weights is implementation-defined.

The *ft:score()* function provides the framework for supporting different scoring mechanisms, but does not dictate the exact scoring mechanism itself. This flexibility is necessary, because vendors are unlikely to agree on the same scoring technique. In fact, scoring for XML is an active area of research (e.g., see [9, 13, 15, 18, 21, 27]), and many vendors view scoring techniques as product differentiators. However, there are two properties that every scoring mechanism must satisfy [32]: (i) the score of a node in the evaluation context must be 0 if and only if the node does not satisfy the full-text condition specified in *FTSelectionWithWeights*. Otherwise, its score must be in the interval (0,1]; (ii) for the nodes in the evaluation context, a higher score value implies a higher degree of relevance to *FTSelectionWithWeights*.

The *ft:score()* function returns a sequence of floating-point numbers, which may occur wherever a number is permitted in other XQuery expressions. This enables the expression of powerful queries such as the one below, which computes the top-10 results for the previous query.

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

The inner FLWOR expression returns the results in descending order by score, and the outer FLWOR expression only returns the top ten of these results.

Our last example illustrates how *FTContainsExpr* and *ft:score()* can be combined to search based on one condition and score based on another one. The expression below selects books that contain *usability* and *analysis*, and these books are scored based on *usability* and *testing*.

```
for $book in //book[. ftcontains "usability" && "analysis"]
let $score := ft:score($book, "usability" weight 0.8 &&
                              "testing" weight 0.2)
return <result score="{$score}"> {$book} </result>
```

3. XQUERY FULL-TEXT IMPLEMENTATION

Numerous strategies exist for implementing XQuery Full-Text – as many strategies as there are for implementing XQuery itself! Possible strategies include extending an existing XQuery engine with native support for the XQuery Full-Text data model and operators; extending an existing full-text search engine to serve as an XQuery Full-Text co-processor; or translating XQuery and XQuery Full-Text into another query language, such as SQL. XQuery Full-Text relies on the *AllMatches* data model that captures words and their positions. Regardless of the implementation strategy chosen, the key implementation problems are representing the *AllMatches* data model, implementing the semantics for each *FTSelection*, and making the word positions used in the input documents accessible to the *AllMatches* data model.

Because new languages benefit from the rapid development of experimental implementations, our strategy was to employ XML and XQuery directly to implement XQuery Full-Text. We first describe key implementation techniques and then their realization in GALATEX.

3.1 General Implementation Techniques

3.1.1 Preprocess Documents & Queries

In the XQuery data model, the text node is the smallest unit representing document content, but in the XQuery Full-Text data model, the smallest unit is a word and its position within a document or phrase. We define an XML value, called *TokenInfo*, to represent a word and its position in an input document or in a search phrase. Two preprocessing steps yield *TokenInfo* values: the text in input documents is tokenized off-line, and the search phrases in a full-text query are tokenized at query evaluation time.

A *TokenInfo* value contains a word and a unique identifier that captures the relative position of the word in a document or in a phrase. When tokenizing document text, a *TokenInfo* may also contain the XML node, sentence, and paragraph that directly contain the word. The DTD for a *TokenInfo* value is below.

```
<!ELEMENT TokenInfo
          (Token, Identifier, Node?, Sentence?, Para?)>
```

As an example, Figure 1 contains a tokenized document in which each word in the text has a corresponding *TokenInfo* identifier, which contains the global position of the word in the document. This information could be augmented with the appropriate node, sentence and paragraph identifiers.

We define abstract functions for pre-processing search phrases and documents. Tokenization of a search phrase is performed by the *getSearchTokenInfo()* function, which takes a search string and returns a sequence of *TokenInfo*s. We explain in Section 3.1.4 how the match-options argument is used during tokenization.

```
getSearchTokenInfo($searchPhrase as xs:string,
   $matchOptions as FTMatchOptions) as TokenInfo*
```

The following abstract functions access tokens and their positions in documents. The *getTokenInfo()* function takes an evaluation context of zero or more element nodes and a search word specified as a *TokenInfo* value and returns all the positions of the word in the given evaluation context. The *getPositions()* function is similar, but restricts the evaluation context to one element node. *getTokenInfo()* and *getPositions()* can both be defined in terms of the *containsPos()* function, which returns true if the given evaluation context contains the given word. The *wordDistance()* function returns the distance between two words given any match options that might affect the *FTWindow* or *FTDistance* primitives.

```
getTokenInfo($evalContext as element()*,
             $searchToken as TokenInfo ) as TokenInfo*
getPositions($node as element(),
             $searchToken as xs:string ) as TokenInfo*
containsPos($node as element()*,
            $searchToken as TokenInfo ) as xs:boolean
wordDistance($token1 as TokenInfo,
             $token2 as TokenInfo,
             $mo as FTMatchOptions ) as xs:integer
```

3.1.2 The AllMatches Data Model

An *AllMatches* value specifies all possible position solutions to a full-text search query and can be viewed as a propositional logic formula in disjunctive normal form (DNF) [1]. We represent instances of the *AllMatches* data model using XML values that conform to the following DTD:

```
<book(1)>
  <author(2)>Millicent(3) Marigold(4)</author>
  <content(5)>
    <p(6)> The(7) usability(8) of(9) software(10) measures(11) how(12) well(13) the(14)
           software(15) provides(16) support(17) for(18) quickly(19) achieving(20)
           specified(21) goals(22).
    </p>
    <p(23)> The(24) users(25) must(26) be(27) and(28) feel(29) well-served(30).
           Software(31) usability(32) is(33) a(34) good(35) measure(36) of(37) that(38).
    </p>
  </content>
  <title(39)>Conquering(40) the(41) systems(42)</title>
</book>
```

Figure 1: XML document fragment with positions

```
<!ELEMENT AllMatches (Match)*>
<!ELEMENT Match (StringInclude|StringExclude)*>
<!ELEMENT StringInclude TokenInfo>
<!ELEMENT StringExclude TokenInfo>
```

Each *Match* in an *AllMatches* corresponds to one of the disjuncts in the DNF formula. Each *StringInclude* in a *Match* corresponds to the proposition that the evaluation context node must contain a word position, and each *StringExclude* specifies that the evaluation context node should not contain a word position.

3.1.3 The FTSelections

Each *FTSelection* function takes one or more *AllMatches* values and returns one *AllMatches*. For example, consider the sample document in Figure 1 annotated with word positions and the following full-text query, which returns those books that contain paragraphs containing words similar to *usability* and *software* case sensitive within ten words of each other:

```
//book[.//p ftcontains ("usability" with stemming) &&
("software" case sensitive) with distance at most
10 words]/title
```

Each full-text query has an associated query evaluation plan of *FTSelections*. Figure 2 contains the plan for the above query. We distinguish between two stages in the evaluation plan. The bottom two levels of the plan construct *AllMatches* values using the position functions described in Section 3.1.1. Once we have the first *AllMatches*, all the other primitives manipulate *AllMatches* only.

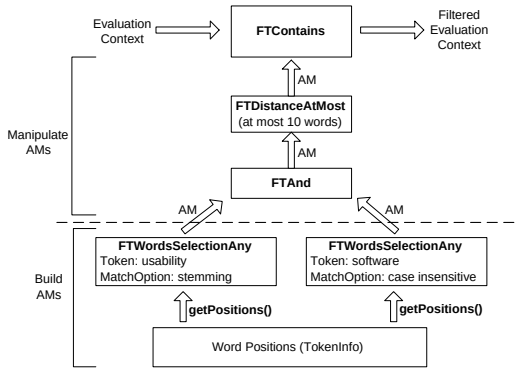


Figure 2: Full-Text XQuery evaluation plan

A variety of primitives build *AllMatches* depending on the query search criteria: a single word (*FTSingleSearchToken*), any word or all words from the set of given phrases (*FTWordsSelectionAnyWord*, *FTWordsSelectionAllWord*), and any or all phrases (*FTWordsSelectionPhrase*, *FTWordsSelectionAny*, *FTWordsSelectionAll*).

In Figure 2, the two *AllMatches* representing the tokens *usability* and *software* become the inputs to the *FTAnd* primitive. The resulting *AllMatches* is given in Figure 3. It contains six possible *Matches*. These *AllMatches* are further filtered by the *FTDistance* primitive. The final *AllMatches* contains only the first, fourth, and sixth *Matches* (see Figure 3). Once those matches are generated, they are passed to *FTContains()*, the top-most node in Figure 2, in order to filter the XML nodes in the evaluation context.

3.1.4 The Match Options

Match options are modifiers that apply to each of the search words. If a match option is not specified explicitly in the query, then its default value is used. The default match options are: *case insensitive, without special characters, without regular expressions, without stemming, without stop words, element content is not ignored, English-language selected, without thesaurus, and diacritics insensitive* [30]. When a match option is specified explicitly in the query, it overrides the default for the phrases to which it applies. The XML representation of match options is:

```
<!ELEMENT FTMatchOptions (FTMatchOption)*>
<!ELEMENT FTMatchOption (FTCaseOption|FTDiacriticsOption|
  FTSpecialCharOption|FTThesaurusOption|FTStemOption|
  FTRegexOption|FTLanguageOption|FTStopWordOption|
  FTIgnoreOption)>
```

The abstract function *applyMatchOption()* applies all match options from *FTMatchOptions* to a list of search tokens and returns the token information for all the modified search words.

```
applyMatchOption($mo as FTMatchOptions,
     $searchToken as xs:string* ) as TokenInfo*
```

3.2 GalaTex Implementation

We describe how these general techniques are realized in our GALATEX architecture depicted in Figure 4. A demonstration of GALATEX and of the XQuery Full-text use cases are available at: *http://www.galaxquery.org/galatex/*. GALATEX is implemented on top of the Galax XQuery engine [11],[3] a complete XQuery implementation that supports functions and modules.

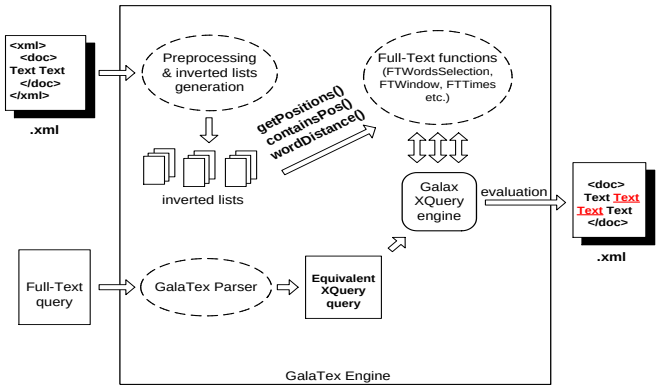


Figure 4: Architecture of GalaTex

In the upper left of Figure 4, GALATEX preprocesses input documents, and for each distinct word, produces one document containing all the positions of that word, represented by *TokenInfo* values. These documents essentially contain *inverted lists*, which map words to their positions. These inverted-list documents are the inputs to *getPositions()* and related functions.

[3] *http://www.galaxquery.org*

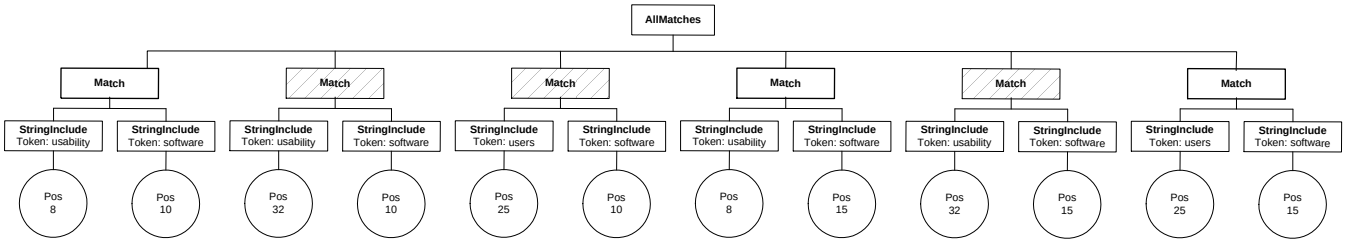


Figure 3: *AllMatches* for "usability" with stemming && "software" case sensitive

In the lower left of Figure 4, GALATEX translates XQuery Full-Text queries into equivalent XQuery queries by mapping each *FTSelection* into a call to the corresponding XQuery function. The XQuery functions themselves (upper right of Figure 4) are implemented in an XQuery library module, where each function implements one *FTSelection* primitive.

On the right of Figure 4, Galax takes the input documents, the translated query, and the library module of XQuery functions, evaluates the translated query, and yields the result as an XML document. The final result contains the relevant XML document fragment in which the search words are highlighted.

The GALATEX library module uses XQuery's optional schema import and validation features, which are supported by Galax. These features are not required by our implementation, but are useful because they guarantee that all *AllMatches* and *FTMatchOptions* values are valid instances of the corresponding types.

3.2.1 Document Preprocessing

The document pre-processing step is done off-line and the result is a set of documents that contain *TokenInfo* values. Our tokenizer assumes that words are delimited by punctuation and whitespace symbols as in English. We chose to implement the *TokenInfo* identifier using Dewey numbering [26]. The Dewey number encodes the depth-first node path from the document root to each node. For each word, the identifier contains the Dewey number of the node containing the word appended with the word's absolute position in the document. For example, in Figure 5(a), the first occurrence of *usability* has identifier 1.3.1.1.4, indicating it is contained in the node with identifier 1.3.1.1 and it is the fourth word in the entire document. For each distinct word identified during tokenization, we create one inverted-list document that contains all of the word's *TokenInfo* values. Figure 5(b) contains the inverted lists for *software*, *usability*, and *users*.

We chose to represent the inverted lists in XML format. The benefit is that all the abstract functions that manipulate positions described in Section 3.1.1 are expressed as XQuery functions operating over XML values. For example, the XQuery implementation of *getTokenInfo()* is given below.

```
declare function
  fts:getTokenInfo( $evalCtx as element()*,
                    $searchToken as fts:TokenInfo)
  as fts:TokenInfo*
{
  for $node in $evalCtx,
      $pos in fts:getPositions($node,
                               $searchToken/@word)
  return
    <fts:TokenInfo word="{$searchToken/@word}"
     prefixPos="{fn:string($pos/@prefixPos)}"
     absPos="{fn:string($pos/@absPos)}"/>
}
```

For each node `$node` in the evaluation context, and for each occurrence of the search word in that node, a *TokenInfo* value is returned. The *getPositions* function accesses the inverted list for `$searchToken` and returns only those positions that are included in `$node`. Testing whether a word position is contained XML node is done in *containsPos()* operator, which compares the integer components of Dewey values hierarchically (e.g., $1.10.1 > 1.9.2$).

3.2.2 Query Parsing & Translation

As shown in Figure 4, the GALATEX parser translates a full-text query into an equivalent XQuery query. This design was chosen to improve portability, to avoid direct impact on the Galax XQuery engine, and to speed implementation. This design also allowed us to implement and test subsets of the XQuery Full-Text specification quite easily while treating the XQuery engine itself as a black box.

Currently, the parser replaces each full-text expression in the original query with the appropriate composition of *FTSelection* function calls. Match options are propagated to the relevant *FTWordsSelection* calls. For example, consider the following full-text query:

```
//book[.//p ftcontains ("usability" with stemming) &&
("software" case sensitive) without stemming with
distance at most 10 words ordered]/title
```

The GALATEX parser produces the following XQuery query:

```
//book[
  ( let $ec_1 := (.//p ) return
  fts:FTContains( $ec_1,
    fts:FTOrdered(
      fts:FTWordDistance(-1, 10,
        fts:FTAnd(
          fts:FTWordsSelectionAny( $ec_1, "usability",
            fts:MO_FTStemOption("with stemming",
            <fts:FTMatchOptions/>), "1"),
          fts:FTWordsSelectionAny( $ec_1, "software",
            fts:MO_FTStemOption("without stemming",
            fts:MO_FTCaseOption("case sensitive",
            <fts:FTMatchOptions/>)), "2")))))
]/title
```

In the translated query, each search string has been replaced with a call to the *FTWordsSelectionAny* function. In addition to the search string, each of these calls also passes the evaluation context (i.e., *//book//p*), the applicable match options, and the position of the search string in the original query. Although it is used in multiple calls, the evaluation context is bound to a variable and is only evaluated once. Note also that the match option *without stemming* has been propagated into each *FTWordsSelectionAny* call (except for *usability* with the explicit *with stemming* override).

The operators '*&&*', '*distance*' and '*ordered*' have been replaced with calls to the full-text functions *FTAnd*, *FTWordDistance* and *FTOrdered*, respectively. The *FTOrdered* function uses the position information for each search string in the query to ensure that words are considered in the order in which they appear in the query.

To do this translation, the parser requires and uses very little knowledge of the XQuery language. In fact, only three tokens were added to our grammar to handle the XQuery language and its overlap with the XQuery Full-Text grammar. These additional tokens:

```
<book(1)>
 <author(1.2)>Millicent(1.2.1.1) Marigold(1.2.1.2)</author>
 <content(1.3)>
  <p(1.3.1)> The(1.3.1.1.3) usability(1.3.1.1.4) of(1.3.1.1.5) software(1.3.1.1.6)
        measures(1.3.1.1.7) how(1.3.1.1.8) well(1.3.1.1.9) the(1.3.1.1.10)
        software(1.3.1.1.11) provides(1.3.1.1.12) support(1.3.1.1.13)
        for(1.3.1.1.14) quickly(1.3.1.1.15) achieving(1.3.1.1.16)
        specified(1.3.1.1.17) goals(1.3.1.1.18).
  </p>
  <p(1.3.2)> The(1.3.2.1.19) users(1.3.2.1.20) must(1.3.2.1.21) be(1.3.2.1.22)
        and(1.3.2.1.23) feel(1.3.2.1.24) well-served(1.3.2.1.25).
        Software(1.3.2.1.26) usability(1.3.2.1.27) is(1.3.2.1.28) a(1.3.2.1.29)
        good(1.3.2.1.30) measure(1.3.2.1.31) of(1.3.2.1.32) that(1.3.2.1.33).
  </p>
 </content>
 <title(1.4)>Conquering(1.4.1.34) the(1.4.1.35) systems(1.4.1.36)</title>
</book>
```

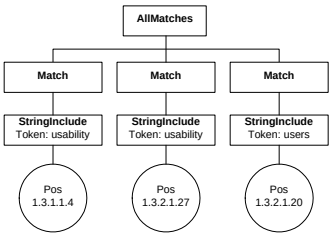
(a) XML document fragment with Dewey positions

Token: "software"	Dewey positions
	1.3.1.1.6 1.3.1.1.11

Token: "usability"	Dewey positions
	1.3.1.1.4 1.3.2.1.27

Token: "users"	Dewey positions
	1.3.2.1.20

(b) Inverted lists



(c) *AllMatches* for "usability" with stemming

Figure 5: Dewey positions and AllMatches example

1. identify the start of XQuery expressions and sub-expressions in order to extract the evaluation context for a full-text expression,
2. identify the return to XQuery from a full-text expression, and
3. disambiguate between parenthesized XQuery expressions and parenthesized full-text expressions in order to identify XQuery expressions embedded within a full-text expression.

Since XQuery code can contain full-text expressions which, in turn, can contain XQuery expressions, arbitrary nesting of the languages is possible and is supported by the parser. In the following example, the result of the embedded XQuery expression is used as a search string.

```
//book[.//p ftcontains
    (//book[./author ftcontains "Marigold"]/title)
    with stemming window at most 15]/title
```

The translated XQuery for the above expression is:

```
//book[
  ( let $ec_1 := ( .//p ) return
    fts:FTContains( $ec_1,
      fts:FTWindow(-1, 15,
        fts:FTWordsSelectionAny( $ec_1,
          (//book[
            ( let $ec_2 := ( ./author ) return
              fts:FTContains( $ec_2,
                fts:FTWordsSelectionAny( $ec_2,
                  "Marigold",
                  <fts:FTMatchOptions/>, "1")))
          ] /title),
          fts:MO_FTStemOption( "with stemming",
            <fts:FTMatchOptions/>), "2"))))
]/title
```

As expected, all of the XQuery-specific code is passed unchanged to the XQuery engine, and each full-text expression has been replaced with an *FTContains* function call (even in the case where one full-text expression is nested inside another). Note that each embedded XQuery expression in the original query must be enclosed in parentheses.

3.2.3 Query Evaluation

3.2.3.1 FTSelections.

A library module of XQuery functions implements the semantics of the *FTSelection* primitives. We return to a simplified version of the query in Section 3.1.3 to illustrate how these functions work.

```
//book[.//p ftcontains ("usability" with stemming) &&
("software" case sensitive) with distance at most
10 words]/title
```

The equivalent XQuery expression generated by the parser is:

```
//book[
( let $ec_1:= ( .//p ) return
  fts:FTContains( $ec_1,
    fts:FTWordDistance(-1, 10,
      fts:FTAnd(
        fts:FTWordsSelectionAny( $ec_1, "usability",
          fts:MO_FTStemOption( "with stemming",
          <fts:FTMatchOptions/>), "1"),
        fts:FTWordsSelectionAny( $ec_1, "software",
          fts:MO_FTCaseOption( "case sensitive",
          <fts:FTMatchOptions/>), "2")))))]/title
```

This translation corresponds to the query plan in Figure 2. We describe this plan "bottom up", beginning with the inner-most function calls to *fts:FTWordsSelectionAny* and ending with the outermost call to *fts:FTContains*. Note that this code is valid, executable XQuery code and not merely a pseudo-code description of a query plan.

The first function, *fts:FTWordsSelectionAny*, constructs the initial *AllMatches*. It calls the *FTSingleSearchToken()* function whose definition in XQuery expression is below.

```
declare function
  fts:FTSingleSearchToken(
     $evalCtx as element()*,
     $searchToken as fts:TokenInfo,
     $matchOptions as fts:FTMatchOptions,
     $queryPos as xs:string ) as fts:AllMatches
{
  <fts:AllMatches>
  {
    for $position in
      fts:getTokenInfo($evalCtx, $searchToken)
    return
      <fts:Match>
        <fts:StringInclude
          queryString="{$searchToken/@word}"
          queryPos="{$queryPos}">{ $position }
        </fts:StringInclude>
      </fts:Match>
  }
  </fts:AllMatches>
}
```

The above function obtains the positions of the search token and constructs one *AllMatches* that contains one *Match* per position. This function uses *getTokenInfo()* described in Section 3.2.1. We defer discussion of match options to Section 3.2.3.2. The last argument to *FTSingleSearchToken()* is (*$queryPos*), which is a variable that contains the relative position of the search word in the full-text query. It is used in conjunction with *FTOrder*. Figure 5(c) shows the *AllMatches* for *usability with stemming*.

The *AllMatches* values constructed for *usability* and *software* are inputs to the *FTAnd* function, which computes the Cartesian product of their *Match*es as follows:

```
declare function
  fts:FTAnd( $allMatches1 as fts:AllMatches,
             $allMatches2 as fts:AllMatches)
  as fts:AllMatches
{
    <fts:AllMatches>
    {
      for $match1 in $allMatches1/fts:Match,
          $match2 in $allMatches2/fts:Match
      return
        <fts:Match>
        { $match1/*, $match2/* }
        </fts:Match>
    }
    </fts:AllMatches>
}
```

This function computes all possible pairs of *Matches* for *usability* and *software* and returns an *AllMatches* value. This value is input to *FTDistance*, which selects those matches that satisfy the distance condition as follows:

```
declare function
  fts:FTWordDistanceAtMost(
    $n as xs:integer,
    $allMatches as fts:AllMatches,
    $matchOptions as fts:FTMatchOptions)
  as fts:AllMatches
{
  <fts:AllMatches>
  {
    for $match in $allMatches/fts:Match
    if fn:empty($match/fts:StringInclude) then
     $match
    else
     let $sorted:= for $si in $match/fts:StringInclude
                   order by $si/fts:TokenInfo/@absPos
                   ascending return $si
     where every $idx in (1 to fn:count($sorted) - 1)
           satisfies fts:wordDistance(
                         $sorted[$idx]/fts:TokenInfo,
                         $sorted[$idx+1]/fts:TokenInfo,
                         $matchOptions) <= $n
     return
       <fts:Match>
         { $match/fts:StringInclude }
         {
          let $sortedStrMatch:=
                        for $si in $match
                        order by $si/*/fts:TokenInfo/@absPos
                        ascending return $si
          for $stringExcl in
            $sortedStrMatch/fts:StringExclude
          where some $stringIncl in
                $sortedStrMatch/fts:StringInclude
                satisfies fts:wordDistance(
                              $stringIncl/fts:TokenInfo,
                              $stringExcl/fts:TokenInfo,
                              $matchOptions) <= $n
          return $stringExcl
         }
       </fts:Match>
  }
  </fts:AllMatches>
}
```

Intuitively, the matches that satisfy *FTDistance* are those for which each pair of adjacent positions satisfy the distance condition. For each of these matches, the included positions and only the excluded positions that fall in the specified distance range are returned.

Finally, *FTContains* filters the evaluation context and returns only those nodes that contain at least one match that satisfies all the inclusion and exclusion constraints.

```
declare function
  fts:FTContains( $evalCtx as element()*,
                  $allMatches as fts:AllMatches)
  as xs:boolean {
   some $node in $evalCtx
    satisfies
      (some $match in $allMatches/fts:Match
       satisfies fts:satisfiesMatch($node, $match))
};

declare function
  fts:satisfiesMatch( $node as element(),
                      $match as fts:Match )
  as xs:boolean {
  ( every $stringInclude in $match/fts:StringInclude
    satisfies fts:containsPos($node,
                $stringInclude/fts:TokenInfo))
  and
  ( every $stringExclude in $match/fts:StringExclude
    satisfies not(fts:containsPos($node,
                $stringExclude/fts:TokenInfo)))
};
```

3.2.3.2 *Match Options.*

In GALATEX, match options are translated by the parser into a set of match option functions implemented in XQuery. A match option has the effect of expanding one search word to a set of words that becomes the new set of search words for the current full-text query. This expansion occurs in *FTSingleSearchToken()*, described in Section 3.2.3.1, which applies *applyMatchOption()* before calling *getTokenInfo()* for the current search word. The function returns the positions of all words that result from applying match options to the current search word. This interface is quite flexible and it allows plugging in any match option implementation.

To implement match options we used the XQuery Functions and Operators defined in [34], in particular, *fn:matches*, *fn:replace*, *fn:lower-case*, *fn:upper-case*. Hence, any XQuery implementation that supports them can be used with GALATEX.

For example, to find the expansion set of a search word when the *case* match option is set to *case insensitive*, we compare for equality the search word with each distinct word from the input document. The list of distinct words is generated in the preprocessing step. Both words are filtered by *fn:lower-case* function as in:

```
let $sToken:= $searchTokens/@word
for $docToken in fn:doc("list_distinct_words.xml")/
  ListDistinctWords/invlist/@word
return
  if (fn:lower-case($docToken)=fn:lower-case($sToken))
  then $docToken
  else ()
```

The same technique works when the *regular expression* match option is active. For the *special character* option we replace the special characters with the following regular expression ".?" and apply the above regular expression technique.

The stemming operation is language specific. GALATEX uses Galax built-in stemmer implementation, which is Porter's English stemmer [22]. The stemmer reduces the English words to their word stems. For example, the word *connections* would be reduced to its stem *connect*.

Stop words are reflected in the implementation of *FTWindow* and *FTDistance* primitives. More precisely, these primitives skip stop words when specified. The remaining match options deal with language specifics and character encoding problems. Their implementation is still underway.

3.3 Full-Text Scoring

In this section, we describe our implementation of a scoring technique in GALATEX. Recall that the specification of XQuery Full-Text [30] does not mandate a specific scoring method. Rather, it defines some requirements on score values based on the relevance of query answers to a full-text expression (see Section 2.2).

The probabilistic relational algebra is a well-established scoring method in Information Retrieval (IR) [23, 14]. This algebra operates on tuples with a score attribute. The score of a tuple represents the probability that a tuple contains a word. A score formula is associated with each algebraic operator which transforms its input tuples scores into output tuples scores. Since each *FTSelection* in our language can be viewed as an algebra operator as illustrated in the query plan in Figure 2, we propose a natural adaptation of the probabilistic scoring method to *AllMatches* and show that it preserves the scoring requirements given in Section 2.

We first add a *score* field to the position structure described in Section 3.1.1 to capture the score of individual positions. This corresponds to adding a score to each entry in the inverted list. Conceptually, the score of an entry represents the probability that the entry contains a given word. Hence, the score value should be a float between 0 and 1. This value can be computed using techniques such as *tf* (term frequency) and *idf* (inverse document frequency) [24].

In order to compute the score of query answers, we associate a score formula with each *FTSelection*. Each formula guarantees that answers will have a score value between 0 and 1. The composition of multiple formulas in a query plan still preserves that property.

- *FTWord* builds an *AllMatches* where each match is assigned the score of the corresponding entry in the input inverted list.

- *FTAnd(*AM_1, AM_2*)*: Given a match m_1 in $\mathtt{AM}_1$ with score s_1, a match m_2 in $\mathtt{AM}_2$ with score s_2 and an output match m_3 that contains m_1 and m_2, the score s_3 of m_3 is:
  $s_3 = s_1 \times s_2$
  This formula is similar to the one used for Boolean AND the probabilistic relational algebra and preserves the fact that the score of tuples has to be a value between 0 and 1.

- *FTOr(*AM_1, AM_2*)*: Given a match m_1 in $\mathtt{AM}_1$ with score s_1, a match m_2 in $\mathtt{AM}_2$ with score s_2 and an output match m_3 that contains m_1 and m_2, the score s_3 of m_3 is:
  $s_3 = 1 - (1 - s1) \times (1 - s2)$
  If m_3 contains only m_1 or m_2, its score will be equal to that of m_1 or m_2.

- *FTDistance* and *FTWindow* accept an *AllMatches* AM as input and return an *AllMatches*. Given a match m in AM with score s, if m satisfies the *FTSelection* then its score s' is:
  $s' = s \times f$ where f is a function associated with the *FTSelection* and evaluates to a value between 0 and 1. For example, the function associated with *FTDistance(AM, dist)* is:
  $distance(m, dist)$ is $f = 1 - s/dist$.

Given a query plan, the final *AllMatches* carries the scores of each match. In order to score a query answer (i.e., an XML node in the evaluation context), we compose the scores of those matches that are contained in that XML node. The composition formula is similar to the one used for *FTOr*. One could use other composition formulas such as *max*.

Note that we do not have specific scoring formulas associated with *FTNegation*, *FTOrder*, *FTScope* and *FTTimes*. In our framework, these operators do not modify the scores of their input tuples. One could devise scoring formulas for each one of them (e.g., *FTTimes* could rely on the number of occurrences of a word for scoring). One interesting direction is approximate matching. For example, if two matches do not satisfy a distance, they might be returned with a lower score.

4. EVALUATION STRATEGIES

Our strategy to implement the XQuery Full-Text language using XML and XQuery is general and expedient, but not very efficient. In this section, we explore improvements to the current query evaluation strategies. We divide this section into improvements on full-text search and improvements on full-text scoring.

4.1 Full-Text Search

Given a query evaluation plan, an obvious optimization would be to push any of the primitives (*FTOrdered*, *FTDistance*, *FTWindow*, *FTScope*, *FTTimes*) as far down in the evaluation tree based on their selectivity. This is akin to pushing selections in the relational algebra. Figure 6(a) shows pushing *FTOrdered*. Another rewriting is short-circuiting the evaluation of *FTOr* by translating into an XQuery "or" (see Figure 6(b)). If one of the resulting branches evaluates to `true`, there is no need to evaluate the other one.

Nodes in an evaluation context might be structurally related, i.e., some might be descendants of others. One could organize nodes in the evaluation context in a way that guarantees that the smallest number of context nodes are checked against *AllMatches*. A node could be marked as an answer if it contains another node that has already been marked as an answer. This would avoid checking that node against *AllMatches*.

Materializing *AllMatches* at each step of a query evaluation tree is one of the main performance bottlenecks when evaluating queries. Pipelining, a well-studied query evaluation method in databases, would reduce the size of materialized intermediate *AllMatches*. This strategy is used in Quark,[4] which implements the TeXQuery language [1]. All our full-text primitives, except *FTTimes*, are non-blocking (i.e., they permit full pipelining of matches in *AllMatches*). *FTTimes* is partially blocking since it needs to materialize a certain number of matches. Given *n* search keywords (*searchToken_1*, ..., *searchToken_N*), the pipeline query evaluation algorithm is as follows:

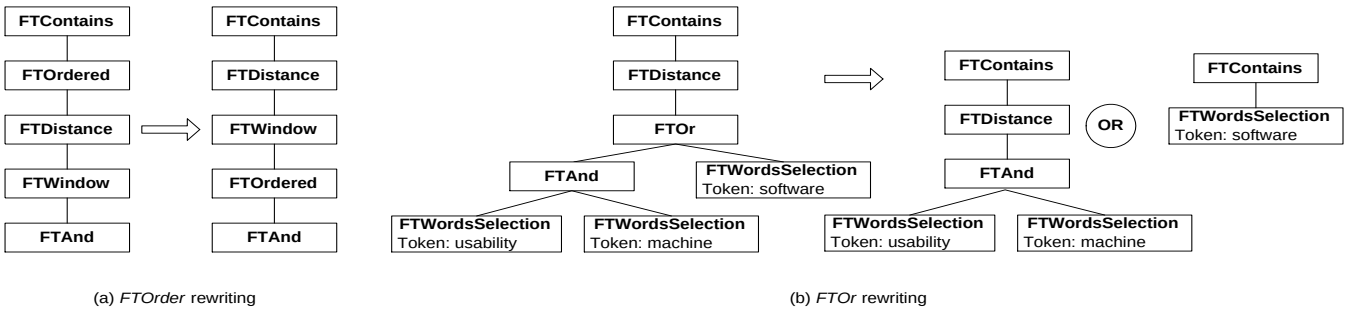
```
$EC <- XQuery/XPath
for $pos1 in getNextPosition_SortMerge(
  unmarked($EC), $searchToken_1)
...
for $posN in getNextPosition_SortMerge(
  unmarked($EC), $searchToken_N)
{
  result <- applyPrimitives($pos1,..., $posN)
  //check ancestor-descendant relationship
  //by computing the least common ancestor (LCA):
  if !empty(result) lca=LCA($pos1,..., $posN)
  if !empty(lca) markNodes($EC, lca)
  //stop condition:
  if succeeded in marking new nodes then break OR
  if allNodesMarked($EC) then break
}
output Boolean result or the marked nodes in $EC
```

We could apply the same pipelining idea to nodes in the evaluation context (i.e., to produce one node at a time). This requires pipelining the execution of the XQuery engine which might not always be possible depending on the engine that is being used. In the last case, not all matches would need to be materialized for a given context node. Figure 7 illustrates pipelining both nodes in the evaluation context and *AllMatches* for the query given below.

```
book[.//p ftcontains "usability" && "software"
     with distance at most 10 words]
```

A more ambitious strategy is fully integrating the XQuery Full Text data model and operators into an XQuery engine. Our strategy to implement each *FTSelection* as an XQuery function permitted rapid prototyping, but unfortunately, semantic functions do

[4] *http://www.cs.cornell.edu/database/Quark*



(a) *FTOrder* rewriting

(b) *FTOr* rewriting

Figure 6: Logical rewritings

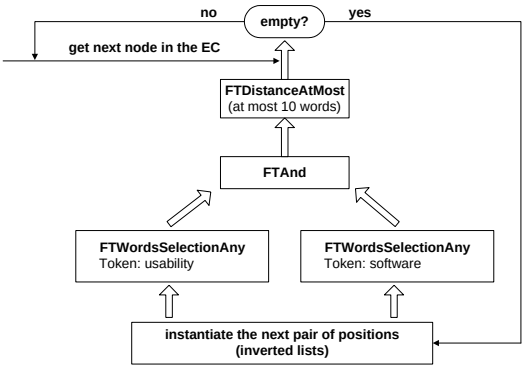


Figure 7: Pipelining Algorithm

not scale, nor do they permit the flexibility required to do query optimization techniques across multiple *FTSelections*. Evolving GALATEX into a scalable implementation is not simple. Taking the step to a fully optimized evaluation strategy requires having fine-grained algebra operators that can be manipulated and composed with an XQuery algebra in order to optimize the integration on queries on structure and text. This requires integrating the *AllMatches* data model with scoring information into the XQuery data model. In particular, one can see that in order to achieve early pruning for top-k queries, we need to be able to push scoring information into an XQuery algebra. How this is achieved is an open question. However, we believe that adapting the probabilistic relational algebra for scoring as explained in Section 3.3 is a first step towards this integration. This would also enable scoring on both structure and content as in [2].

4.2 Full-Text Scoring

In Section 3.3, we showed that in the current GALATEX implementation, in order to score an answer (i.e., a node in the evaluation context), we produce an *AllMatches* that carries a score for each one of the matches that it contains. This means that we need to materialize all the matches in the *AllMatches* produced by a full-text evaluation plan. In the previous section, we discussed a pipelining approach to reduce the need to materialize all intermediate results. This conflicts with the need to materialize *AllMatches* to compute final answer scores. In order to implement scoring and still benefit from pipelining, we could estimate upper-bounds on the scores of those matches that have not been materialized and on the number of matches that a node might have in order to compute its scores without having to materialize all its matches.

5. RELATED WORK AND CONCLUSION

In database and IR research, several languages have been proposed for processing XML data on structure and text. The main focus was put on extending existing XML query languages with full-text search. However, unlike XQuery Full-Text, previous solutions explore only a few full-text search primitives at a time (e.g., Boolean keyword retrieval [12, 35], keyword similarity [8, 27], proximity distance [5], relevance ranking [4, 7, 8, 13, 18, 27]). Further, previous techniques did not provide a seamless integration with XQuery which permits querying both structure and text. More importantly, these approaches did not develop a fully composable model for full-text search the way *AllMatches* does for example.

Various ranking models have been proposed for XML in the IR literature, including the vector space model [25] and probabilistic models [23, 5]. These models provide a systematic way to compute the relevance of a document to a query. Recently, some of these models have been adapted to incorporate document structure into account when ranking query answers. This has been the main focus of the proposals submitted to INEX. In particular, XIRQL [13] and XXL [27] extend the probabilistic model while JuruXML [7] and ELIXIR [8] extend the vector space model.

Table 1 classifies existing XML full-text search proposals according to available search primitives and scoring techniques. Many IR engines for XML extend existing XML query engines as in the second column. We can see in this table that GALATEX fills a gap in the space of expressiveness of query languages for XML. Some of these languages incorporate explicit or implicit textual and context (element names) similarity operators used in the ranking mechanism. Most of them have decided to include limited XPath navigation in the input query and allow SQL-like queries (ELIXIR, XXL, XIRQL). Other languages have considered a more simple and intuitive query syntax by either specifying the query as an XML fragment (JuruXML) or in a Google-like style through a list of pairs: element name and keyword (XSearch). There are different approaches on the granularity of query output. XXL and ELIXIR are able to return document fragments. On the contrary, XIRQL and JuruXML focus more on relevance-oriented search and let the engine decide what nodes to return.

In this paper, we discussed the implementation of the XQuery Full-Text language [30], an extension of XQuery language [28] that provides fully composable full-text search primitives. The TeXQuery language [1] is the main precursor of XQuery Full-Text [30]. We presented GALATEX the first conformant implementation of XQuery Full-Text that is able to query XML documents both on structure and text content. GALATEX uses XML and XQuery to implement XQuery Full text, which permits implementation on top of any existing XQuery engine. One interesting direction is to explore the use of an existing IR engine to implement some *FTSelec-*

IR engines	XML query engine	Search primitives	Weighting on query terms	Similarity operator	Scoring
XQuery Full-Text [30] (GALATEX)	XQuery	phrase matching, Boolean connectives, order specifier, proximity distance, no. occurrences, match options (stemming, regular expressions, stop words, case sensitive)	yes	implicit	probabilistic model or vector space model
XIRQL [13] (HyREX)	XQL	phrase matching, Boolean connectives, *$sounds_like$* operator	yes (query terms and document terms)	textual and context	probabilistic model
Flexible XML Search [27] (XXL)	XML-QL	phrase matching, limited Boolean connectives, *LIKE* operator	no	textual and context (similarity join)	probabilistic model
ELIXIR [8]	XML-QL	phrase matching, limited Boolean connectives	no	textual (similarity join)	vector space model
JuruXML [7]	Juru	phrase matching, limited Boolean connectives (negation)	no	implicit, textual and context	vector space model

Table 1: Classification of existing IR engines for XML

tion and benefit from IR scoring techniques for relevance ranking.

6. REFERENCES

[1] S. Amer-Yahia, C. Botev, J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. WWW 2004.

[2] S. Amer-Yahia, L. Lakshmanan, S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. SIGMOD 2004.

[3] S. Amer-Yahia, M. Fernandez, D. Srivastava, Y. Xu. Phrase Matching in XML. VLDB 2003.

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

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

[6] 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.

[7] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, A. Soffer. Searching XML Documents via XML Fragments. In SIGIR 2003.

[8] T. T. Chinenyanga, N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data. WebDB 2001.

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

[10] E. Curtmola, S. Amer-Yahia, P. Brown, M. Fernández. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. WWW 2005 (poster).

[11] M. Fernández, J. Siméon, B. Choi, A. Marian, G. Sur. Implementing XQuery 1.0: The Galax Experience. VLDB 2003.

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

[13] N. Fuhr, K. Grossjohann. XIRQL: An Extension of XQL for Information Retrieval. SIGIR 2000.

[14] N. Fuhr, T. Rölleke. A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems. ACM TOIS 15(1), 1997.

[15] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[16] Health Level Seven. http://www.hl7.org/.

[17] V. Hristidis, L. Gravano, Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. VLDB 2003.

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

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

[20] Library of Congress. http://xml.house.gov/.

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

[22] Porter M.F. An algorithm for suffix stripping. 1980.

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

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

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

[26] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and querying ordered XML using a relational database system. SIGMOD 2002.

[27] A. Theobald, G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. EDBT 2002.

[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 1.0 and XPath 2.0 Full-Text. W3C Working Draft 09 July 2004. http://www.w3.org/TR/2004/WD-xquery-full-text-20040709/

[31] The World Wide Web Consortium. XQuery and XPath Full-Text Use Cases. W3C Working Draft 09 July 2004. http://www.w3.org/TR/xmlquery-full-text-use-cases/

[32] The World Wide Web Consortium. XQuery and XPath Full-Text Requirements. W3C Working Draft 02 May 2003. http://www.w3.org/TR/xmlquery-full-text-requirements/.

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

[34] 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/.

[35] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001.