Full-Text Querying in XML

A Little Bit of Standards and Lot's o' Research

Sihem Amer-Yahia

AT&T Labs–Research

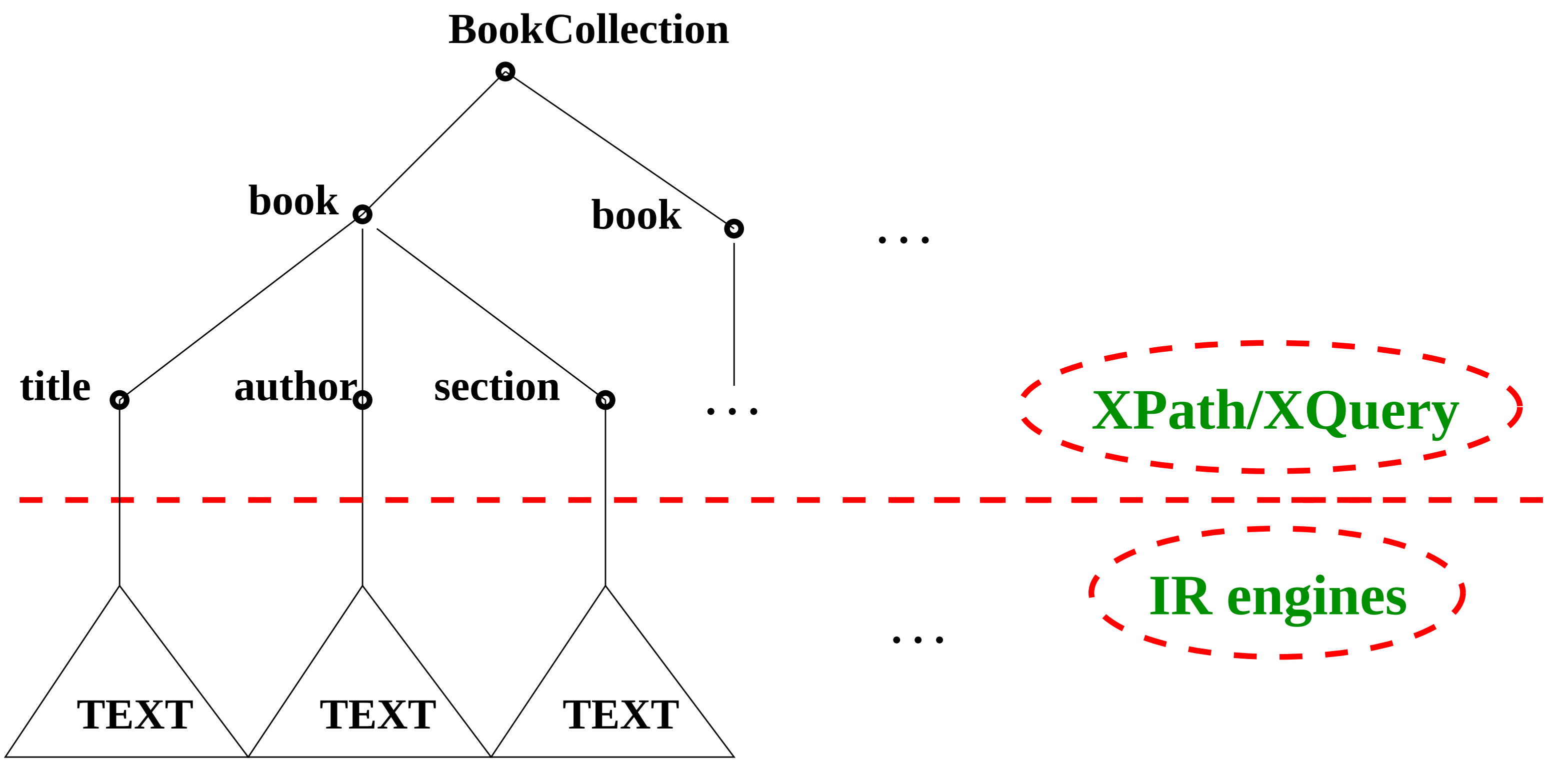
EDBT Summer School – September 6^{th}-10^{th}, 2004

Motivation

1. XML is able to represent a mix of structured and unstructured (text) information.
2. Examples of XML repositories are: IEEE INEX (INitiative for Evaluation of XML retrieval) data collection, Shakespeare's plays, DBLP and the Library of Congress collection.
3. Existing query languages for XML are very limited when querying text.
4. A lot of activity around the topic of extending XML query languages with full-text search capabilities: INEX (IR effort), W3C (DB effort). Bring together the two communities.

Outline

1. **Full-Text Search in XML:**
   - Motivation and Related Work.
   - TeXQuery and the Standards.
   - Demonstration.
2. **Research in XML Full-Text Search:**
   - PIX: Phrase matching In XML.
   - FleXPath: Approximate Matching on Structure and Text.
   - Open research problems.
3. **Bibliography**



Querying an XML document in DB and in IR

Related Work

Querying / Languages/ Tools	STRUCTURE	TEXT	SCORING
IR Engines (Google,XIRQL, ELIXIR, XXL, JuruXML, ...)	* Limited path expressions * Dynamic context evaluation * Structure used mainly for scoring purposes	* Powerful text search * Not fully composable * "incomplete" * Efficient indices and algorithms	* Powerful scoring using well−established measures (TF*IDF) * Limited use of structure
XPATH 2.0 XQuery 1.0 XSLT 1.0	* Powerful tree navigation primitives * Powerful "return" clause	* Limited sub−string matching (starts−with, contains, ...) * Coarse data model	* None
Full−Text Search in XML	* Leverage power of XPath and XQuery to specify search context and return clause	* Fine−grained data model (at the level of words) * Powerfull and fully−composable full−text search primitives * Efficient query evaluation for both structure and text	* Scoring on both text and structure * Extend TF*IDF to account for structure

Full-Text Search in XML

Context expression: defines nodes where search occurs: *e.g.*, book chapters.

Return expression: defines document fragments that are returned to users: *e.g.*, book title and authors.

Search expression: defines Full-Text search conditions: *e.g.*, Boolean, proximity, stemming.

Score expression: defines an expression that might be used to score returned fragments.

Full-Text Queries

```
<book id="1000">
  <author>Elina Rose</author>
  <content>
    <p> The usability of software measures how
        well the software provides support for
        quickly achieving specified goals. </p>
    <p> The users must be and feel well-served. </p>
  </content>
</book>
```

- `//book [./content ftcontains "software" && "usability" with stems && ! "Rose"]`
- `//book [./chapter ftcontains "usability" && ("software" || "goals") distance 12]`

XQuery in a Nutshell

- Functional language.
- Input/Output: sequence of items (atomic types, elements, attributes, processing instructions, comments, ...).
- Fully compositional.
- Variable binding.
- XPath core navigation language.
- Element construction (return clause).

XQuery FLWOR Expression

Find the title and price of books on usability and sort books from the cheapest to the most expensive:

```
for $item in //books/book
let $pval := $item/metadata/price
where   fn:contains($item//content,"usability")
order by $pval ascending
return <result>
        {$item/title}
        <price>
          {$pval}
        </price>
      </result>
```

Limited sub-string operations: *fn:start-with()*, *fn:end-with()*.

Full-Text Search Design Goals

- Identify *basic Full-Text search primitives* natural to querying XML.
- Primitives should be *composable* with each other to express arbitrarily complex Full-Text conditions.
- Seamlessly *integrate regular XQuery with Full-Text search* to query over both structured and full-text data. Non-trivial because structured XML queries operate on XML nodes, while Full-Text queries operate on keyword search tokens and their positions *within* XML nodes.
- *Avoid any extension to the XPath and XQuery data model.*
- Define *ranking* in order to support threshold and topK queries.

Alternative Solutions: Functions

```
word-distance(contains($n,"usability")
              and
              contains($n,"software"),10)
```

- ”contains” returns Boolean values. Not enough to compute distance.
- Extra information about search tokens and their positions needs to be ”carried around” with the Boolean value.
- **Problem:** Fundamental extension to the XQuery data model, violating design goals.

Alternative Solutions: Sublanguage

```
contains($n,
     "usability and software distance 10 words")
```

- Isolate Full-Text search expresson in a single `contains` function as in SQL/MM, an extension to SQL.
- No extension to XQuery data model is needed.
- **Problem:** Full-Text search specified in an uninterpreted string that is opaque to the rest of the XQuery language.
- **Solution:** Make string conform to a well-defined grammar and define its semantics.

TeXQuery in a Nutshell

- Provides a set of powerful Full-Text search primitives called **FTSelection**s.
- FTSelections are *fully composable*.
- Relies on a formal data model called **FullMatch**.
- Permits scoring and ranking.

TeXQuery Primitives

- **FTContainsExpr::= ContextExpr ”ftcontains” FTSelection** returns true if at least one node in ContextExpr satisfies FTSelection.
- **FTScoreExpr::= ContextExpr ”ftscore” FTWeightedSelection** returns a sequence of scores. Provides access to fine-grained ranking (e.g., threshold and top-k.)

FTContainsExpr: FTSelection

FTContainsExpr ::= ContextExpr "ftcontains" FTSelection

```
FTSelection  ::= FTStringSelection |
                 FTAndConnective |
                 FTOrConnective |
                 FTNegation |
                 FTMildNegation |
                 FTOrderSelection |
                 FTScopeSelection |
                 FTDistanceSelection |
                 FTWindowSelection |
                 FTTimesSelection |
                 FTSelection (FTContextModifier)*
```

FTContainsExpr: FTContextModifier

FTSelection ::= FTSelection (FTContextModifier)*

FTContextModifier defines the FTS environment, which can modify the operational semantics of FTSelection such as stemming, stop-words, diacritics and case.

```
FTContextModifier  ::= FTCaseCtxMod |
                       FTDiacriticsCtxMod |
                       FTSpecialCharCtxMod |
                       FTStemCtxMod |
                       FTThesaurusCtxMod |
                       FTStopWordCtxSpec |
                       FTLanguageCtxMod |
                       FTRegExCtxMod |
                       FTIgnoreCtxMod
```

FTContextModifiers: Grammar

```
FTThesaurusCtxMod ::= "with"? "thesaurus" Expr
           | "without" "thesaurus"

FTStopWordsCtxMod ::= "with" "additional"?
           "stopwords" Expr ?
           | "without" "stopwords" Expr?

FTLanguageCtxMod ::= "language" Expr
```

FTContainsExpr Examples

```
books//title [. ftcontains ("usability") case
sensitive with thesaurus "synonyms" ]

books//content [. ftcontains ("usability" &&
 "software") with stopwords window at most 3 ]

books//title [. ftcontains ("Utilisation" language
"French" with stems && "software") ]

books//content [. ftcontains ("usability" ||
"web-testing") with special characters ]
```

Integration with XQuery

- **Simple Example:**

```
for $book in
  books/book  ftcontains "usability" with stems &&
                         "software" && !"Rose"
return <hit>{$book}</hit>
```

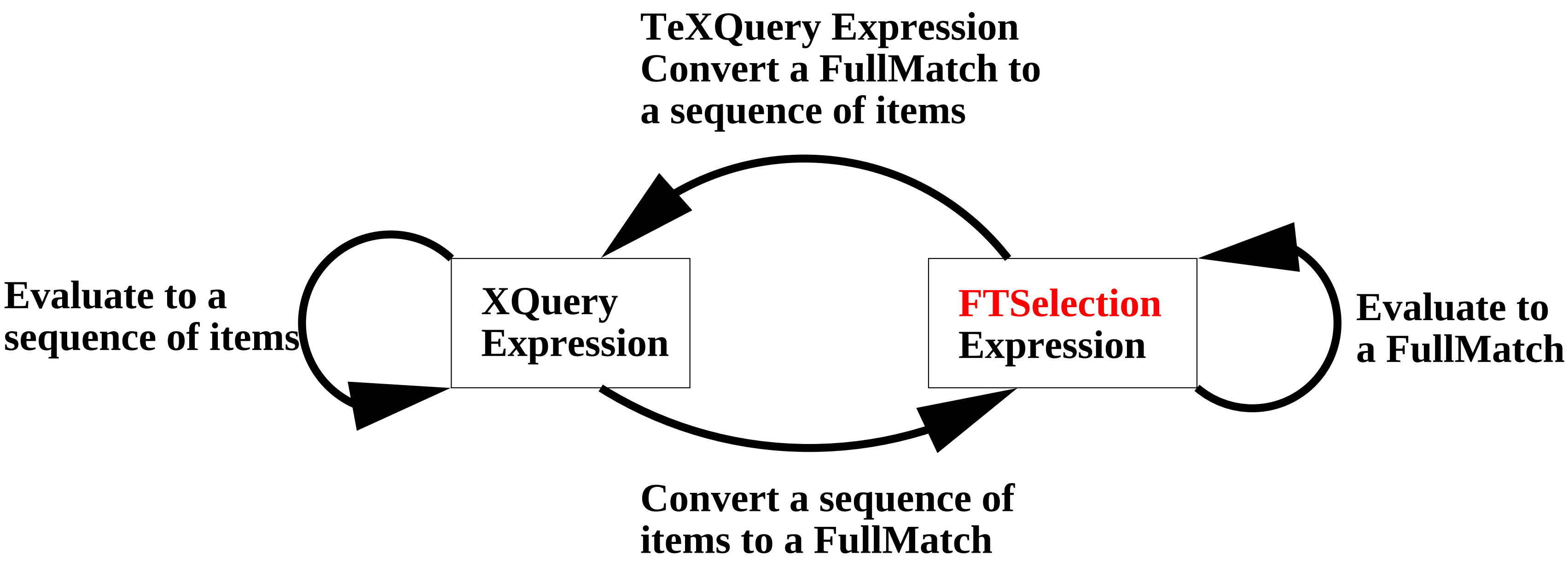
- **Top-K Example:**

```
for $hit at $i in
  for $book in  books//section ftcontains "usability"
  let $score := $book  ftscore "software" weight 0.7
  order by $score descending
  return <hit>{$book}<score>{$score}</score></hit>
where $i < 20
return {$hit}
```

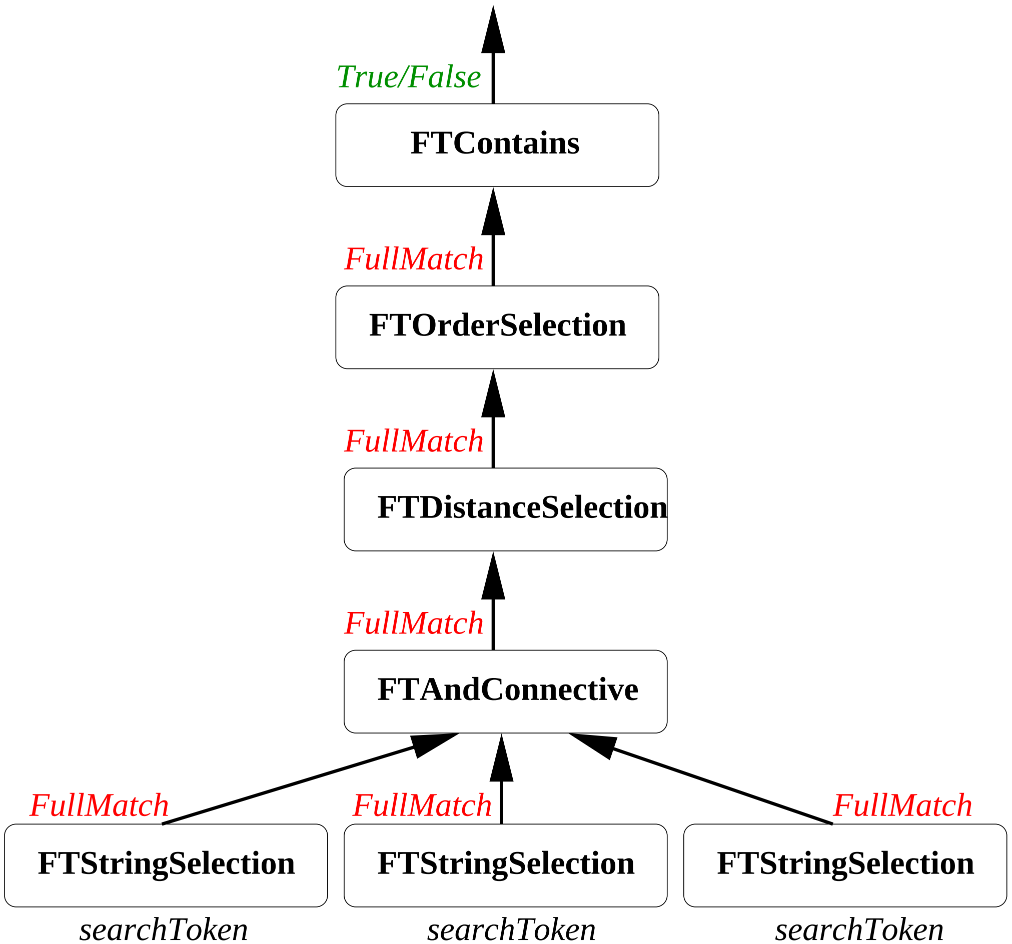
TeXQuery Data Model

- XQuery expressions take sequence(s) of nodes as input and evaluate to a sequence of nodes.
- FTSelection takes FullMatch(es)as input, and evaluates to a FullMatch in the FTS data model.
- FullMatch captures linguistic token positions, and other information required for full composability of FTSelections.

XQuery and TeXQuery Composability



Query Evaluation Tree



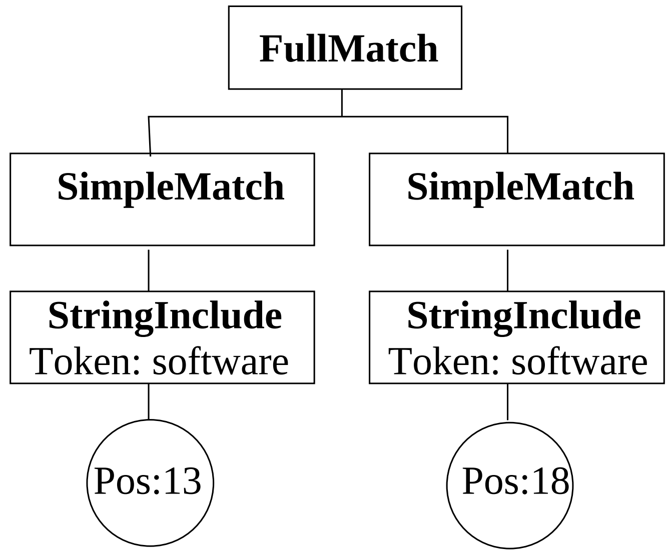
TeXQuery Example

```
<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) be(31) and(32)
     feel(33) well-served(34).
    </p(35)>
  </content(36)>
</book(37)>
```

//book ftcontains "software" && "usability" with stems

"software"

```
<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)> ...
```

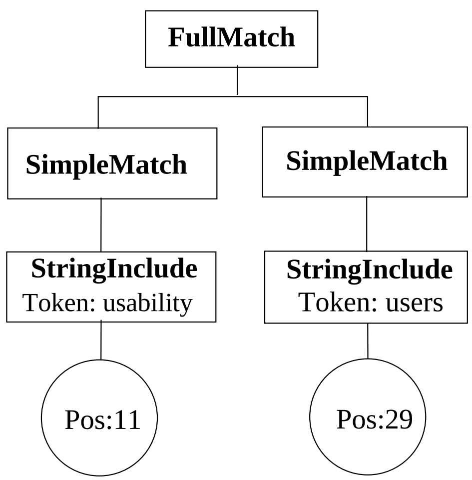


FTStringSelection

```
function fts:FTStringSelection (
          $searchContext as node(),
          $ctxModifiers as fts:FTctxModifiers,
          $searchToken as fts:TokenInfo,
          $queryPos as xs:integer) as fts:FullMatch
{  <FullMatch>
      let $token_pos := fts:getTokenInfo($searchContext,
                                  $matchOptions,$searchToken)
       for $pos in $token_pos
       return <match> <stringInclude queryPos="$queryPos"
                       queryString="$searchToken/@word">
                       $pos
                      </stringInclude>
              </match>
   </FullMatch>
}
```

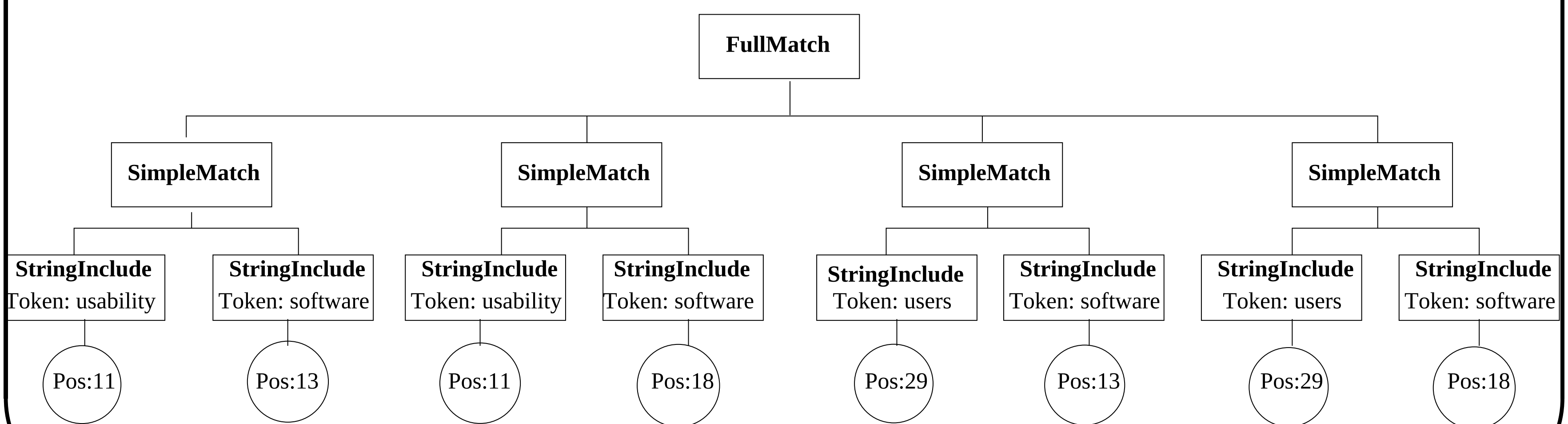
”usability” with stems

```
<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) be(31) and(32)
 feel(33) well-served(34).
</p(35)> ...
```



”usability” with stems && ”software”

```
<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) be(31) and(32)
 feel(33) well-served(34).
</p(35)> ...
```



FTAndConnective

```
function fts:FTAndConnective (
              $fullMatch1 as fts:FullMatch,
              $fullMatch2 as fts:FullMatch)
         as fts:FullMatch
{  <FullMatch>
    { for $sm1 in $fullMatch1/match,
          $sm2 in $fullMatch2/match
        return
           <match>
             $sm1/* $sm2/*
           </match>
    }
   </FullMatch>
}
```

Related Work

- SQL/MM extends SQL with primitives on text, images and spatial data. Boolean keyword retrieval [FKM00],[NDM00]. Keyword similarity [CK01],[XXL],[XIRQL:FG01]. Proximity distance [Inquery:sigir95],[SQL/MM:sigrecord01]. Relevance ranking [XQueryIR:webdb02],[FG00],[HTK00],[TW00]. Dynamic context [SKW01],[XRank:GSB+03],[TIX:AYJ03]
- All support only a few FT search primitives at a time and none develops a fully compositional model for FT search.

A Quick Summary of W3C Effort

- Full-Text Task Force (FTTF) started in Fall 2002 to extend XQuery with full-text search capabilities: IBM, Microsoft, Oracle, the US Library of Congress.
- FTTF documents published on February 14, 2004 (public comments are welcome!):
  http://www.w3.org/TR/xmlquery-full-text-use-cases/
  http://www.w3.org/TR/xmlquery-full-text-requirements/
- XQuery Full-Text highly influenced by TeXQuery.
- Published a working draft describing the syntax and semantics of the XQuery Full-Text on July 9, 2004 at:
  http://www.w3.org/TR/xquery-full-text/

FTTF Use Cases Document

http://www.w3.org/TR/xmlquery-full-text-use-cases/

- Use Case ”ELEMENT”: Words and Phrases
- Use Case ”WILDCARD”: Word Wildcard
- Use Case ”STEMMING”: Word Stemming
- Use Case ”THESAURUS”: Thesauri, Dictionaries, and Taxonomies
- Use Case ”STOP-WORD”: Ignoring and Overriding Stop Words
- Use Case ”BOOLEAN”: Or, And, Not
- Use Case ”DISTANCE”: Distance (Proximity, Window)
- Use Case ”IGNORE”: Ignoring Markup
- Use Case ”COMPOSABILITY”: Full-Text and XQuery
- Use Case ”SCORE”: Scoring and ranking

XQuery Full-Text Demo

The GalaTex Prototype

PART 2: Research in XML Full-Text

- In IR:
  - Ranking for XML.
  - Querying both structure and text: returned results granularity.
- In DB:
  - Indices and algorithms for phrase matching.
  - Approximate querying of both structure and text: algorithms to evaluate top-K queries efficiently.
- Implementation on top of an XPath/XQuery engine and an IR engine.

PIX: Phrase Matching in XML

```
<section>
  <title> Web site Testing </title>
  <p> Software <footnote> The word software
      designates programs and tools </footnote>
      usability measures how well the
      software provides support to users.</p>
</section>
```

- Two kinds of markup: tags or annotations. Affects contiguity of words in phrase.
- Ignore annotation `<footnote>`.

  book//section[. `ftcontains "software usability" case insensitive ignore content footnote`]

PIX Problem Statement

- Given a (pre-processed) XML document, and
- Proximity query specified by:
  - context node tags C
  - list of phrase words $W = [w_1, \ldots, w_q]$
  - ignore-tag tags T
  - ignore-annot tags A
  - proximity threshold K
- **Identify all (context node, witness list) pairs in document**

PIX Contributions

- Dynamic specification (i.e., at query time) of phrase to match & markup to ignore.
- Inverted indices on words & XML tags built off line.
- Phrase (contiguous words in order)/proximity (within k words and tags), while ignoring markup during query evaluation.
- Implementation is fully integrated into XQuery: combines structure matching with phrase matching.
- Carry extensive experiments.

PIX Indices

(Start, End) numbering of XML elements and text.

```
<section [1,24]>
  <p [2,23]> Software [3] <footnote [4,12] >
      The [5] word [6] software [7] designates [8]
      programs [9] and [10] tools [11]
      </footnote> usability [13] measures [14]
      how [15] well [16] the [17] software [18]
      provides [19] support [20] to [21] users [22].
  </p>
</section>
```

INL and PIX Algorithms

```
<section [1,24]>
  <p [2,23]> Software [3] <footnote [4,12] >
      The [5] word [6] software [7] designates [8]
      programs [9] and [10] tools [11]
      </footnote> usability [13] measures [14]
      how [15] well [16] the [17] software [18]
      provides [19] support [20] to [21]users [22].</p>
</section>
```

$L_{section}$	$L_{footnote}$	$L_{software}$	$L_{usability}$
[1,24]	[4,12]	[3,3]	[13,13]
		[7,7]	[22,22]
		[18,18]	

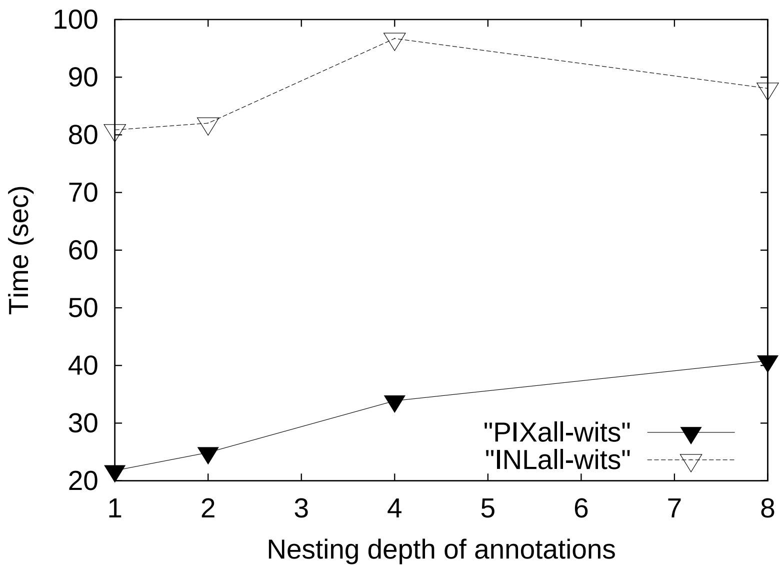
- INL: Build B-tree on Start and End positions for probing.
- PIX: Sort-merge akin to structural joins.

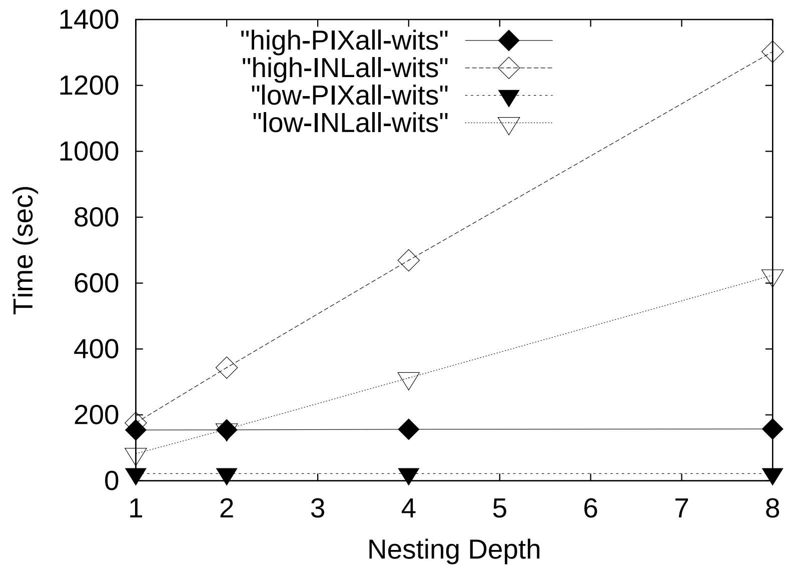
Experiments: Applicability of Known Results

- No context nesting, no ignored markup
  - akin to relational joins
- $\mid L_{w_1} \mid \ll \mid L_{w_2} \mid$
  - INL is substantially better
- $\mid L_{w_1} \mid \sim \mid L_{w_2} \mid$
  - PIX is superior

Experiments: Exploring Variability

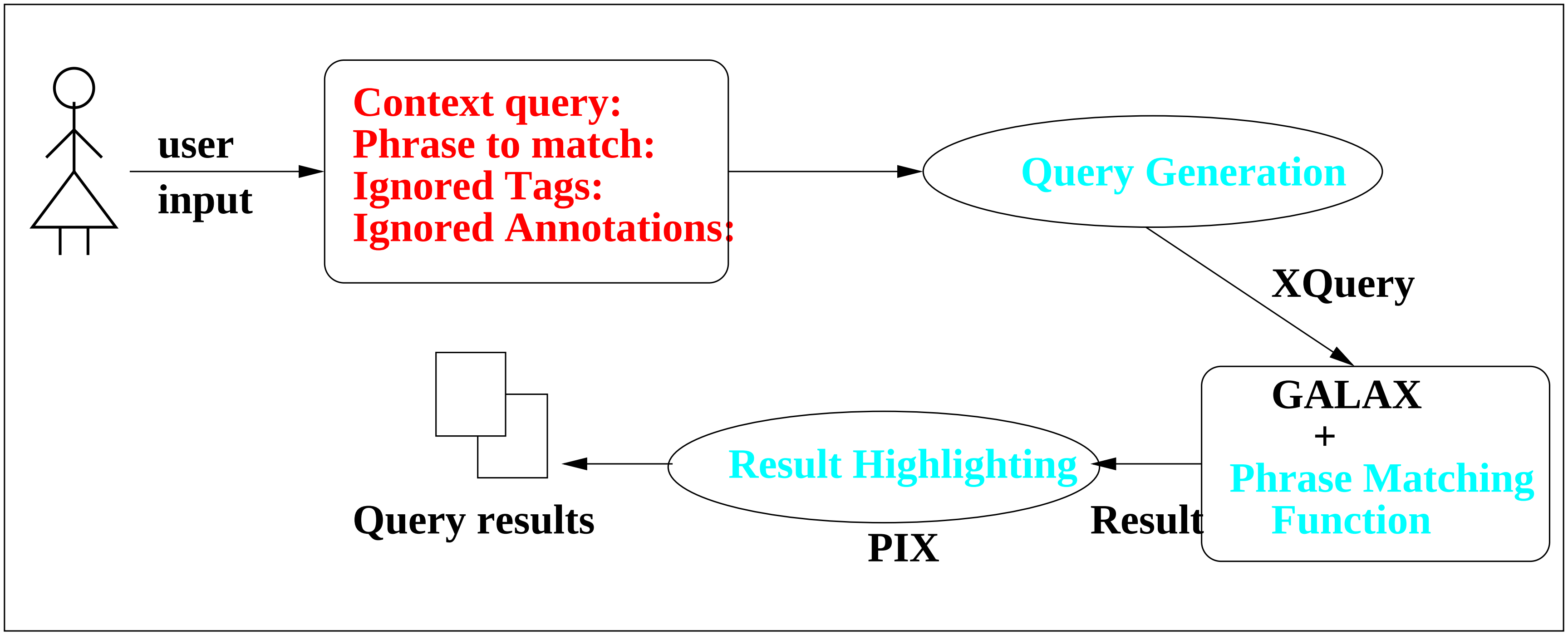
Vary nesting of context nodes and annotations





- PIX is independent of nesting depth, INL increases linearly
  - repeated index probing for nested context nodes

PIX Architecture



FleXPath Motivation

1. *Two compelling paradigms for querying XML documents:*
   - *Database-style query languages*: XPath provides powerful primitives to navigate in document structure.
   - *IR-style querying*: Keyword/full-text search provides powerful search primitives at the fine level of element and attribute content.
2. *Study query evaluation and scoring challenges that arise when combining these two paradigms.*

FleXPath Basic Ideas

- **Facts:** XPath has exact match semantics. Keyword search is based on approximate matching.
- **Goal:** Leverage XPath in specifying the search context and, at the same time, not suffer from the consequences of the exact match semantics of XPath.
- **Idea:** Treat queries on structure as a *template* and look for answers that *best match* the template and the full-text search.
- **Consequence:** If input document satisfies XPath expression *exactly*, it is returned. If input document satisfies expression *partially*, it might be returned with a lower score.

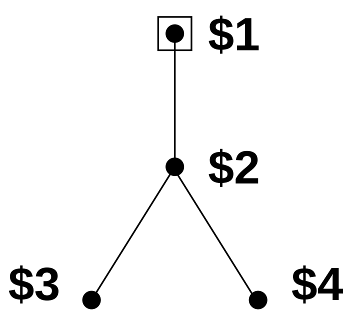
FleXPath Contributions

1. Formalize *query relaxation on structure* that is relevant to keyword search.
2. Develop a query semantics that consistently extends classical semantics of queries without full-text search.
3. Define primitive operators to span the space of relaxations.
4. Study properties of ranking schemes that combine structure and text and propose ranking schemes.
5. Develop efficient algorithms for answering top-K queries.
6. Carry experimental evaluation.

Queries in FleXPath

XPath expressions where a predicate might call the *fn:contains* function which looks for occurrences of specified keywords. In general, *fn:contains* can be any TeXQuery expression.

//article [./section [./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]

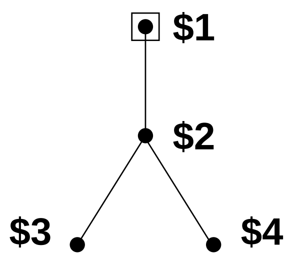


($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

O1

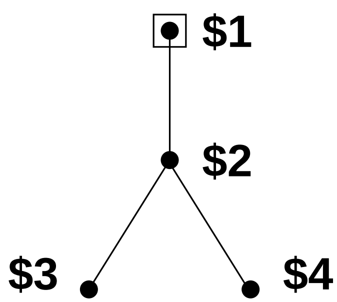
Relaxation Example

*//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]*



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

*//article [./section[./algorithm and
./paragraph and . contains ("XML" and "streaming")]]*

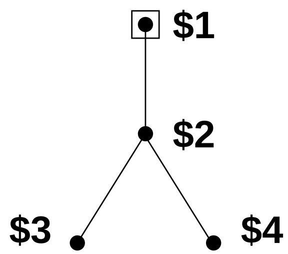


($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($2, "XML" and "streaming")

Q2

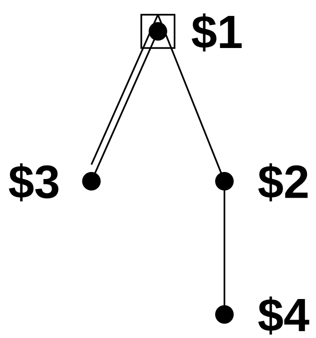
Relaxation Example

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

//article [.//algorithm and ./section [./paragraph [
. contains ("XML" and "streaming")]]]

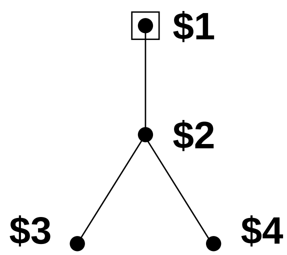


($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains($4, "XML" and "streaming")

O3

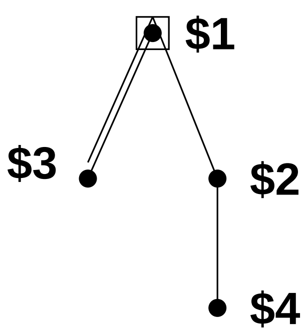
Relaxation Example

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

//article [.//algorithm and ./section [./paragraph and
. contains ("XML" and "streaming")]]

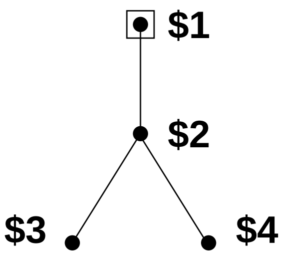


($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains($2, "XML" and "streaming")

O4

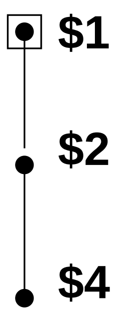
Relaxation Example

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

//article [./section [./paragraph and
. contains ("XML" and "streaming")]]

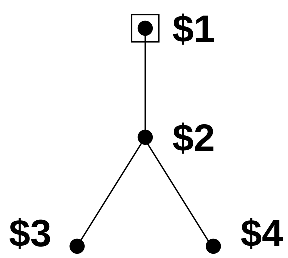


($1.tag = article) &
($2.tag = section) &
($4.tag = paragraph) &
contains ($2, "XML" and "streaming")

O5

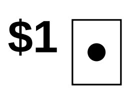
Relaxation Example

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

//article [. contains ("XML" and "streaming")]

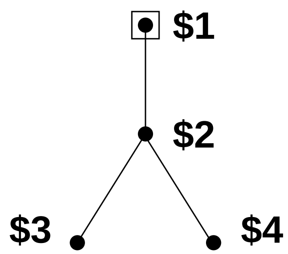


($1.tag = article) &
contains ($1, "XML" and "streaming")

O6

Logical Expression of $Q1$

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

$pc(\$1, \$2) \ \wedge \ pc(\$2, \$3) \ \wedge \ pc(\$2, \$4) \ \wedge$ \$1.tag = article $\wedge$
\$2.tag = section $\wedge$ \$3.tag = algorithm $\wedge$ \$4.tag = paragraph $\wedge$
$contains$(\$4, “XML” and “streaming”).

Our approach for Relaxation

1. Compute *query closure* using inference rules below:

$$pc(\$x, \$y) \vdash ad(\$x, \$y)$$

$$ad(\$x, \$y),\ ad(\$y, \$z) \vdash ad(\$x, \$z)$$

$$ad(\$x, \$y),\ contains(\$y, \texttt{FTExp}) \vdash contains(\$x, \texttt{FTExp})$$

2. Drop predicates.

3. Compute *query core* (unique).

Relaxations Definitions

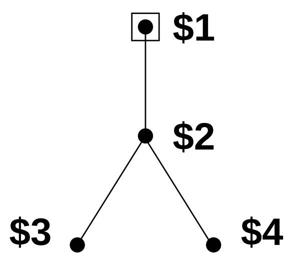
Let $Q = (T, F)$ be a TPQ, C be its closure, and $S \subset C$ be a set of structural predicates.

Definition 1 [Structural Relaxation] *A* structural relaxation *of Q is any query $C - S$, provided (i) $C - S$ is not equivalent to C and (ii) the core of $C - S$ is a tree pattern query.* ■

Definition 2 [*contains*-Relaxation] *Let $contains(\$i, \texttt{FTExp})$ be a predicate in F, such that $\$i$ is not the root of T. Then $Q' = (T, F')$, where F' is identical to F except $contains(\$i, \texttt{FTExp})$ is replaced by $contains(\$j, \texttt{FTExp})$, where $\$j$ is an ancestor of $\$i$ in T, is a* contains-relaxation *of Q.* ■

Query Closure of $Q1$

//article [./section[./algorithm and
./paragraph [. contains ("XML" and "streaming")]]]



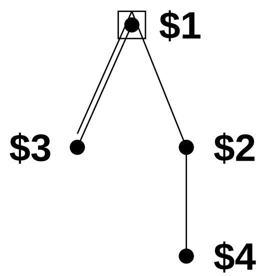
($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains ($4, "XML" and "streaming")

$pc(\$1, \$2) \wedge pc(\$2, \$3) \wedge pc(\$2, \$4) \wedge \$1.tag =$ `article` $\wedge\ \$2.tag =$ `section` $\wedge\ \$3.tag =$ `algorithm` $\wedge\ \$4.tag =$ `paragraph` $\wedge$ $contains(\$4,$ “XML” and “streaming”$)$ $\wedge\ ad(\$1, \$2) \wedge ad(\$2, \$3) \wedge ad(\$2, \$4) \wedge ad(\$1, \$3) \wedge ad(\$1, \$4) \wedge$ $contains(\$2,$ “XML” and “streaming”$)$ $\wedge$ $contains(\$1,$ “XML” and “streaming”$)$.

Relax $Q1$ to $Q3$

Structural relaxations *must* be defined using the closure of a TPQ. $Q3$ can be obtained only from closure of $Q1$.

//article [.//algorithm and ./section [./paragraph [
. contains ("XML" and "streaming")]]]



($1.tag = article) &
($2.tag = section) &
($3.tag = algorithm) &
($4.tag = paragraph) &
contains($4, "XML" and "streaming")

$pc(\$1, \$2) \ \wedge \ pc(\$2, \$4) \ \wedge \ ad(\$1, \$3) \ \wedge \ \$1.tag = article \ \wedge$
$\$2.tag = section \ \wedge \ \$3.tag = algorithm \ \wedge$
$\$4.tag = paragraph \wedge \ contains(\$4,$ "XML" and "streaming").

Core of $C - \{pc(\$2, \$3), ad(\$2, \$3)\}$.

Spanning Relaxations

- 1. Axis Generalization (γ)
  2. Leaf Deletion (λ)
  3. Subtree Promotion (σ)
  4. "contains" Promotion (κ)
- One could consider more relaxations that can be represented in our framework.

Theorem 1 [Soundness and Completeness] : *Let Q be a TPQ. Every query that is obtained by applying a composition of one or more of the operators $\gamma, \lambda, \sigma, \kappa$ applied to Q is a valid structural or* contains *relaxation. Every valid relaxation of Q can be obtained by finitely many applications of these operators to Q.* ■

Ranking Schemes

1. **Structural score:** reflects how well an answer structurally matches the original query.
2. **Keyword score:** reflects the relevance of an answer to the full-text expression.
3. **Answer score:** reflects the relevance of a query answer to the original query. Obtained using a computable arithmetic function that combines the structural and the keyword scores.
4. Different from existing content and structure ranking schemes in IR that rely on pre-specified XML fragments.

Properties of Ranking Schemes

Theorem 2 [Good Ranking Schemes] : *Let Q be a TPQ, w_Q a function that associates a weight with each predicate in Q, and f be an aggregate function. Suppose the score of each answer t to query Q or one of its relaxations is computed by the ranking scheme: $f(\{\{w_Q(p_1), ..., w_Q(p_k)\}\})$, where $p_1, ..., p_k$ are the predicates satisfied by the answer t and $\{\{...\}\}$ denotes a multiset. Then the ranking scheme is order invariant.* ■

Aggregate function used may be arbitrary – i.e., distributive (like sum), algebraic (like average), or holistic (like median).

A Specific Ranking Scheme

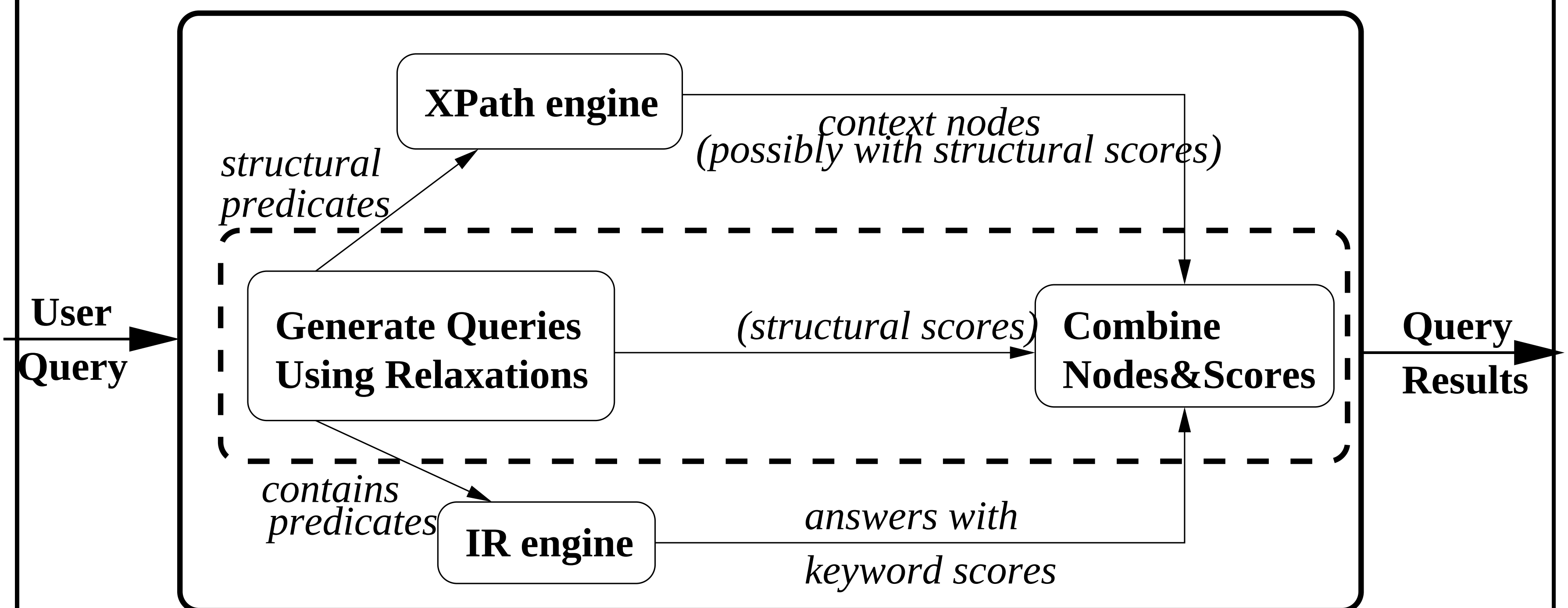
1. *Predicate penalty* associated with each predicate p in C measures how much context an answer loses by not satisfying that predicate.

2. Penalty of relaxing pc to ad:
   $[\#_{pc}(t_i, t_j)/\#_{ad}(t_i, t_j)] \times w_Q(pc(\$i, \$j))$

3. Penalty of dropping $ad(\$i, \$j)$:
   $[\#_{ad}(t_i, t_j)/(\#(t_i) \times \#(t_j))] \times w_Q(ad(\$i, \$j))$

4. Penalty of dropping $contains(\$i, \texttt{FTExp})$:
   $[\#_{contains}(\$i, \texttt{FTExp})/\#_{contains}(\$l, \texttt{FTExp})] \times w_Q(contains(\$i, \texttt{FTExp}))$

FleXPath General Architecture



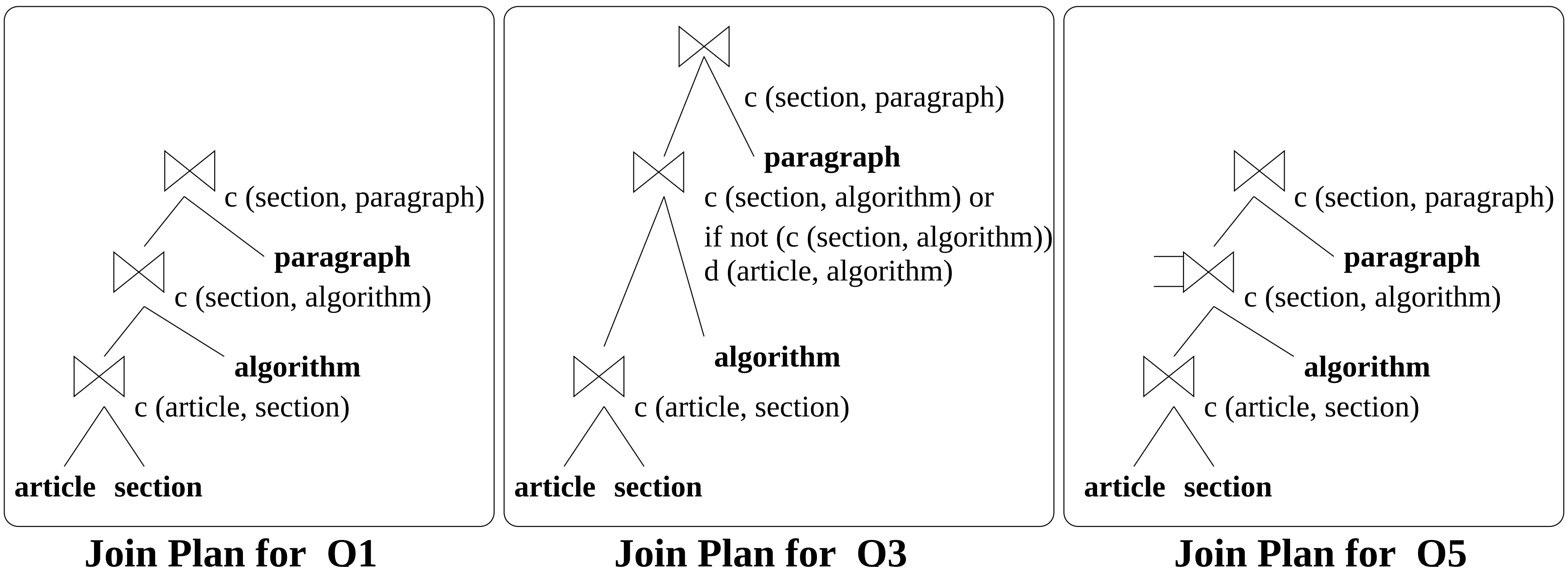
Algorithms: Challenges

1. Leverage off-the-shelf XPath and IR engines.
2. Use any ranking scheme. In practice, keyword and structural scores may result from different engines.
3. Optimize repeated computation due to relaxations.
4. Optimize cost of (re)sorting answers due to scoring.
5. Optimize number of intermediate query answers to produce top-K.
6. All our algorithms assume that structural conditions are evaluated before any *contains* predicate.

Three Algorithms

1. Rewriting-based algorithm (DPO). Relaxations are sorted on penalty. Evaluates one query per relaxations. Stops query evaluation when number of answer exceeds K.
2. Selectivity-based algorithm (SSO). Uses selectivity estimates to decide which relaxations to encode in a query in order to generate at least K answers before sending that query (only once) to the XPath and IR engines.
3. Hybrid: Join evaluation requires sorting intermediate answers on their ids while pruning intermediate answers requires their sorting on scores. Fundamental tension between these two sort orders.

Join Plans for Q_1, Q_3 and Q_5



Related Work

- In IR, CAS approaches include ELIXIR, XIRQL and JuruXML. Allow limited XPath queries ad focus on a vague matching of limited XPath predicates and on designing specific indices to score document fragments.
- Relaxations on structure defined by [Delobel and Rousset'02]: unfold node, delete node, propagate condition at a node to its parent, [Schlieder'02] and [Fuhr'00]: generalize datatypes, ontologies on elements, edit distance on paths, delete node, insert intermediate nodes and rename node.
- SSO is similar to works that use statistical information to map top-K relational queries into selection predicates.

Summary of TeXQuery, PIX and FleXPath

1. Language for full-text search in XML based on a formal semantics (WWW 2004 paper and SIGMOD 2004 demonstration).

2. Efficient indices and algorithms to evaluate one FTSelection: phrase matching in XML (SIGMOD 2003 demonstration).

3. Formal framework for *approximating queries on structure* in order to view queries on structure as a template for keyword search (SIGMOD 2004 paper) and efficient algorithms for answering top-K queries.

Open Research Problems

- **Indices and algorithms:** IR techniques to evaluate other FTSelections, and context modifiers efficiently.
- **Scoring and ranking:** Generalize TF*IDF measure from IR to account for document structure.
- **Combining structure and text:** Evaluate structure-first or keyword-first or interleave and its impact on scoring.
- **Pipelining of FullMatch evaluation**: materialize only necessary matches – impact on scoring.
- **Top-K algorithms**: Computing approximate answers motivates the need for adaptive query evaluation strategies.

PIX/TeXQuery References

- TeXQuery language and semantics presented at WWW 2004.
- TeXQuery demo presented at SIGMOD 2004 (built on top of Quark). Demo today built on top of Galax.
- http://www.research.att.com/~sihem/TeXQuery/
- PIX demo presented at ICDE 2003 and SIGMOD 2003.
- PIX algorithm published in VLDB 2003. Demo today built on top of Galax.
- http://www.research.att.com/~sihem/PIX/

Bibliography

- S. Amer-Yahia, C. Botev, J. Robie and J. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. WWW 2004.
- S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. EDBT 2002.
- S. Amer-Yahia, Mary Fernàndez, Divesh Srivastava and Yu Xu. Phrase Matching in XML. VLDB 2003.
- J. Bosak. The plays of Shakespeare in XML. `http://www.oasis-open.org/cover/bosakShakespeare200.html`
- J. M. Bremer and M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. WebDB 2002.
- E. W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. SIGIR 1995.
- N. Bruno et al. Top-K Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. ACM TODS, 27(2), 2002.
- D. Carmel et al. Searching XML Documents via XML Fragments (JuruXML). SIGIR 2003.

• T. Chinenyanga and N. Kushmerick. An expressive and efficient language for XML information retrieval. Ed. R. Baeza-Yates et al., ASIST 2002 (Special issue on XML and IR).

• T. T. Chinenyanga and N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data (ELIXIR). WebDB 2001.

• W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. SIGMOD 1998.

• S. Cohen et al. XSEarch: A Semantic Search Engine for XML. VLDB 2003.

• E. Damiani et al. The APPROXML Tool Demonstration. EDBT 2002.

• E. Damiani and L. Tanca: Blind Queries to XML Data. DEXA 2000.

• DBLP in XML.
`http://dblp.uni-trier.de/xml/`

• C. Delobel and M.C. Rousset. A Uniform Approach for Querying Large Tree-structured Data through a Mediated Schema. International Workshop on Foundations of Models for Information Integration (FMII-2001).

• D. Florescu, I. Manolescu, and D. Kossmann. Integrating keyword search into XML query processing. WWW 2000.

• N. Fuhr and K. Grossjohann. XIRQL: An extension of XQL for information retrieval. SIGIR 2000 Workshop on XML and IR.

• N. Fuhr, T. Rlleke. HySpirit a Probabilistic Inference Engine for Hypermedia Re- trieval in Large Databases. EDBT 1998.

• L. Guo et al. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

• Y. Hayashi, J. Tomita, and G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. SIGIR 2000 Workshop on XML and IR.

• M. J. Healey et al. Precise environmental searches: EnviroDaemon with hierarchical information search. Environmental Quality Management, 1999.

• V. Hristidis et al. PREFER: A system for the Effi cient Execution Of Multiparametric Ranked Queries. SIGMOD 2001.

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

• Y. Kanza et al. Queries with incomplete answers over semistructured data. PODS 1999.

• Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. PODS 2001.

• P. Kilpelainen. Tree Matching Problems with Applications to Structured Text Databases. PhD thesis, University of Helsinki, Finland, November 1992.

- The Library of Congress.
`http://lcweb.loc.gov/crsinfo/xml/`

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

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

- G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, New York, 1983.

- T. Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. EDBT 2002.

- D. Shasha, K. Zhang, and J. Wang. TreeSearch: Searching among unordered trees, February 2001. http://cs.nyu.edu/cs/faculty/shasha/papers/treesearch.html

- Sigmod Record in XML.
`http://www.acm.org/sigmod/record/xml/`

- A. Theobald and G. Weikum. Adding relevance to XML. WebDB 2000.

- H. Turtle, B. Croft. Inference Networks for Document Retrieval. SIGIR 1990.

• J. E. Wol , H. Fl orke, and A. B. Cremers. XPRES: a ranking approach to retrieval on structured documents. Technical Report IAI-TR-99-12, University of Bonn, July 1999.

• The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text. W3C Working Draft.
`http://www.w3.org/TR/xquery-full-text/`

• The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text Use Cases. W3C Working Draft.
`http://www.w3.org/TR/xmlquery-full-text-use-cases/`

• The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text Requirements. W3C Working Draft.
`http://www.w3.org/TR/xmlquery-full-text-requirements/`