Flexible and Efficient XML Search with Complex Full-Text Predicates

Sihem Amer-Yahia
AT&T Labs Research
180 Park Ave.
Florham Park, NJ
sihem@research.att.com

Emiran Curtmola*
UC San Diego
9500 Gilman Dr.
La Jolla, CA 92093
ecurtmola@cs.ucsd.edu

Alin Deutsch*
UC San Diego
9500 Gilman Dr.
La Jolla, CA 92093
deutsch@cs.ucsd.edu

ABSTRACT

Recently, there has been extensive research that generated a wealth of new XML full-text query languages, ranging from simple Boolean search to combining sophisticated proximity and order predicates on keywords. While computing least common ancestors of query terms was proposed for efficient evaluation of conjunctive keyword queries by exploiting the document structure, no such solution was developed to evaluate complex full-text queries. We present efficient evaluation algorithms based on a formalization of full-text XML queries in terms of keyword patterns and an algebra which manipulates pattern matches. Our algebra captures most existing languages and their varying semantics and our algorithms combine relational query evaluation techniques with the exploitation of document structure to process queries with complex full-text predicates. We show how scoring can be incorporated into our framework without compromising the algorithms complexity. Our experiments show that considering element nesting dramatically improves the performance of queries with complex full-text predicates.

1. INTRODUCTION

The recent increase in the number of XML repositories [19, 22] has motivated extensive work on designing languages for XML full-text search [10, 13, 14, 21, 25, 26, 28], designing scoring methods [2, 8, 10, 14, 16] and developing efficient query evaluation algorithms [1, 17, 24, 29]. However, previous efforts were either tightly connected to the idiosyncrasies of each language or focused on the common denominator of existing languages, namely conjunctions of terms, thus not addressing optimization in the presence of complex full-text search predicates. We propose a unified approach to the optimization of a large class of rich XML query languages with full-text predicates. The approach is based on translating them to an algebra which supports rewriting optimizations, score computation and efficient evaluation, by combining relational set-at-a-time processing techniques with XML-specific exploitation of the nested document structure.

Complex full-text predicates are needed to meet the expressivity demands of increasingly sophisticated XML search applications such as digital libraries [22], and more recently, the Initiative for the Evaluation of XML retrieval methods (INEX) [19], a TREC-like effort for XML. The challenges to a unified treatment of XML full-text search are posed by the variety of expressive powers and semantics. Queries range from simple keyword search to a complex combination of full-text predicates and operate on the textual content of leaf elements in the XML document tree, returning elements satisfying the predicates. Figure 1 shows a fragment of an XML document extracted from the Library of Congress collection [22]. A typical query would look for

> *all elements containing the terms* `Jefferson` *and* `education` *within a window of 10 words, with* `Jefferson` *ordered before* `education` [26, 28].

Existing languages may return the most specific [17, 21] or all elements satisfying full-text predicates, possibly filtered by a user-specified structured query [14, 25, 26, 28]. For a given answer full-text predicates are checked against occurrences of query terms (also called *matches*) that belong to that answer. Predicates are usually interpreted in one of two ways. Under *binding semantics*, the same match within an answer must satisfy all query predicates. Under *existential semantics*, query predicates may be satisfied by different term matches within an answer.

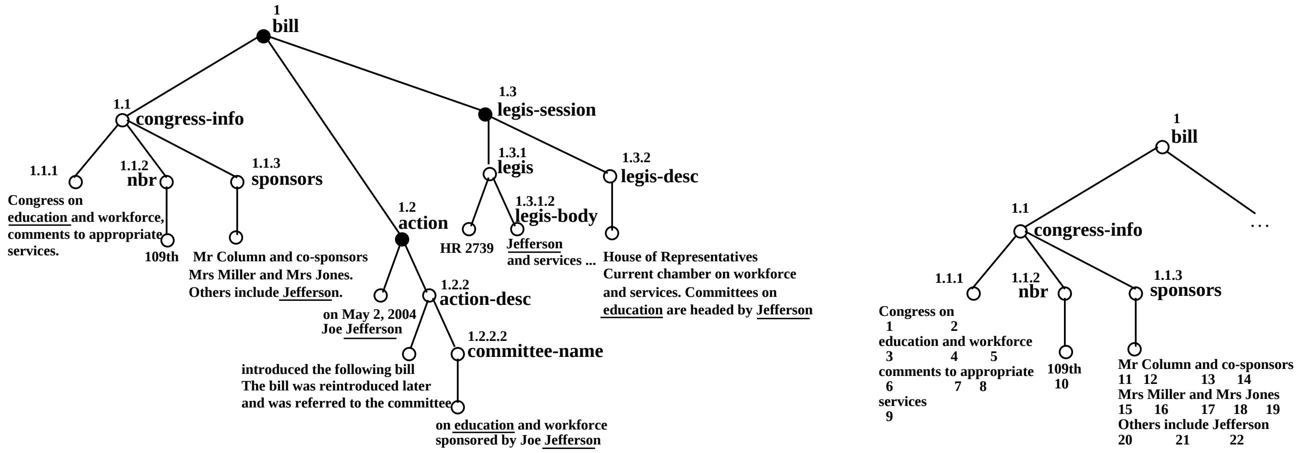
Full-text predicates pose a challenge to efficient evaluation as well. Under both semantics, due to element nesting, evaluating predicates on each element independently may result in redundant work, as matches nested within an element must be considered again when evaluating the predicates on its ancestors. The challenge is to compute the smallest necessary number of elements and matches and use element nesting to infer qualifying answers. This problem may seem simple at first since such solutions have been proposed in the past for conjunctive keyword queries (no predicates) by computing lowest common ancestors (LCAs) [17, 21, 24, 29]. However, due to the interplay between expressive full-text predicates which necessitate to keep track of matches, and flexible query semantics, a direct application of existing solutions would not suffice.

EXAMPLE 1.1. *Consider the query given above on the document on Figure 1. Under binding semantics of the predicates, the query returns the empty answer, since whenever a match for* `Jefferson` *precedes* `education`*, they are too far apart.*

A reasonable relaxation of the query involves the more flexible existential semantics, which yields the nodes shown in solid

[*] Supported by NSF CAREER award IIS-0347968.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.



(a) Example XML Document

(b) Example Term Positions

Figure 1: XML Document for Example 1.1

circles in Figure 1(a). Each node is selected because it contains one match of the query terms which satisfies the order predicate and another match which satisfies the window. Prior work [17, 21, 24, 29] focuses on queries which specify only conjunctions of terms, for which the most specific answers are sought. These are the LCAs of the elements in which individual query terms occur. In our example, the set of LCAs is `congress-info`, `action`, `committee-name`, `legis-session`, `legis-desc` *and* `bill`. *The natural extension to predicate support corresponds to computing with each LCA only the matches which are not nested within a descendant LCA, as this compressed representation suffices to infer all matches. In our example, this means that we only keep matches contained in* `action` *which are not in* `committee-name`. *If we now apply the window predicate to the LCA set, we obtain* `committee-name` *and* `legis-desc` *since none of the other LCA nodes contains a valid pair of matches. The order predicate will then result in the empty answer. We obtain the same empty result even if the order in which the predicates are applied is reversed.*

In both cases the results are incomplete if compared with the correct answer (the solid circles). This is due to the fact that matches are only kept within their most specific element.

One could envision two query evaluation strategies which compensate for the problem illustrated by Example 1.1. First, one could keep with each LCA all query term matches in its subtree, even those nested within descendant LCAs. This would lead to redundant computation and defeat the purpose of working with LCAs in the first place. An alternative fix would simply apply all full-text predicates simultaneously to each node. However, we would like to devise query evaluation algorithms that work regardless of the order in which predicates are applied, guaranteeing full compositionality of the predicates with conditions requiring the match of term conjunctions, disjunctions and negations. We therefore face the challenge of integrating efficient LCA computation with efficient match manipulation (violated by the first fix), while guaranteeing such properties as commutativity, reordering and free compositionality for the full-text predicates (violated by the second fix).

In addition, ranking in our context must account for different query semantics and guarantee the highest scores for the most relevant answers. Existing relevance ranking methods discussed in related work, do not take query predicates into account. We propose to account for full-text predicates when scoring query answers and show that element scores can be computed incrementally from their descendant elements without compromising query performance.

In summary, this paper makes the following contributions: **1.** We introduce a formalization of XML full-text queries in terms of keyword pattern matches and we present an algebra called XFT which constructs and manipulates pattern matches using conjunction, disjunction and difference operators, as well as selection with complex full-text predicates.

XFT permits flexible query semantics by supporting binding and existential interpretations of full-text predicates. Thus, most existing full-text languages can be expressed in XFT, which enables a uniform treatment of their evaluation and optimization problems.[1] The XFT operators are freely composable, enabling query rewriting based on algebraic equivalences in the spirit of relational algebra optimization. Finally, XFT can be seamlessly integrated with algebras for structured XML search such as [1, 27], thereby enabling the optimization of queries which combine structured and full-text predicates [5, 10, 14, 25, 26, 28, 30].

2. To show the feasibility of efficient evaluation of XFT expressions, we devise an algorithm called SCU (for Smallest Containing Unit), which minimizes the number of elements and matches it needs to compute at each operator in order to evaluate all query answers. Our algorithm combines relational query evaluation techniques with the stack-based exploitation of element nesting. While stack algorithms have been widely employed for LCA computation [4, 17, 21, 29], they do not straightforwardly apply to our setting; they violate our compositionality requirement by consuming input sorted according to the preorder traversal of the document and producing postorder-sorted output. The SCU algorithm takes a novel approach which transforms postorder-sorted input into postorder-sorted output. This is made possible by the off-line generation of inverted lists.

3. XFT operators enable the definition of scoring methods that account for the satisfaction of query predicates and can thus incorporate flexible query semantics (binding and

[1] Translation into XFT also yields semantics specifications which are significantly more concise than the standard-provided ones (see Section 2.4.)

existential). We show how element scores can be computed incrementally from their descendants, without compromising the complexity of evaluating the XFT operators.

4. We run experiments that study the performance of the SCU algorithm and show that accounting for element nesting to evaluate complex full-text predicates improves query response time by several orders of magnitude when compared to a naive evaluation of the algebra. Our performance results compare favorably to GalaTex [12], the reference implementation of XQuery Full-Text [28].

The paper is organized as follows. Section 2 presents the XFT algebra, its formal semantics, some algebraic equivalences, and the translation of XQFT [28] and NEXI [26] into XFT. It also shows how XFT can incorporate answer scoring. Section 3 describes the XFT evaluation algorithms, while their implementation and performance evaluation are reported in Section 4. We present related work in Section 5 and conclude in Section 6.

2. THE XFT ALGEBRA

Formalization of full-text languages. We start from the observation that typical XML full-text languages (we shall call their family the *XQFT class* after the most expressive among them, namely XQuery Full-Text [28]) have a common characteristic: their semantics can be formalized in terms of the individual matches of keyword patterns in the input document, possibly filtering them using predicates.

Patterns and Matches. In XQFT-class languages, an expression's principal role is to specify *patterns* which are tuples of terms to be simultaneously matched against the XML document. A singleton term (k) is a pattern whose *matches* are the positions at which the term appears in the document. A term position uniquely identifies a term match and preserves order and proximity information between terms in the document. For instance, the term `education` in Figure 1(b) appears at positions 3 and 45 and 67 further in the document. The matches of a pattern $(k_1, k_2, \ldots, k_n)$ are tuples $(m_1, m_2, \ldots, m_n)$ where each m_i is a match of term k_i. Since some full-text languages allow negation (see XQFT in Section 2.4), an expression may in general specify an *inclusion-exclusion pattern pair* such that each pair pp has two attributes: $pp.I$ holding one pattern and $pp.E$ holding a set of patterns. Intuitively, such expressions specify nodes with matches of the inclusion pattern $pp.I$ but no matches of the exclusion patterns in $pp.E$. Due to the presence of disjunction, queries may return *sets* of inclusion-exclusion pattern pairs. For example, the XQFT expression

`(Jefferson && education || committee) && ¬Thomas`

specifies the pattern pairs

I	E
(Jefferson, education)	{(Thomas)}
(committee)	{(Thomas)}

The matching table. For any XQFT-class language, we define the semantics of a query Q to be a nested table $[\![Q]\!](N, P, M)$ (called a *matching table*), where N is an XML element node, P is a pattern, and M is a set of matches. $[\![Q]\!]$ collects the matches of all patterns specified by Q as follows: For each inclusion-exclusion pattern pair pp of Q and each XML element N such that N contains no matches of $pp.E$'s patterns, the set M of matches of $pp.I$ contained in N is non-empty, and N satisfies the predicates appearing in the query, $[\![Q]\!]$ contains a tuple t where $t.N = N$, $t.P = pp.I$, and $t.M = M$.

EXAMPLE 2.1. *The matching table for a simple query asking for all elements containing the term* `education` *on the document in Figure 1(a) is*

N(ode)	*P(attern)*	*M(atches)*
bill	(education)	$\{(3), (45), (67)\}$
congress-info	(education)	$\{(3)\}$
action	(education)	$\{(45)\}$
action-desc	(education)	$\{(45)\}$
committee-name	(education)	$\{(45)\}$
legis-session	(education)	$\{(67)\}$
legis-desc	(education)	$\{(67)\}$

XFT *algebra.* We designed the XFT algebra to construct matching tables for queries in all XQFT-class languages, thus enabling a uniform treatment of their evaluation and optimization problems. XFT facilitates rewriting optimizations and lends itself to efficient evaluation using algorithms that combine relational query evaluation techniques with the exploitation of document structure to process XML queries with complex full-text predicates. Section 2.1 defines the XFT data model and operators which are inspired from relational algebra. In Section 2.2, we show relational-style algebraic rewritings which can benefit optimizations. Section 2.4 shows translations into XFT for the XQFT [28] and the NEXI [26] languages. Section 2.3 explains how XFT permits scoring of query answers under flexible query semantics.

2.1 XFT Operators

The XFT operators manipulate matching tables, composing them into new tables or filtering their tuples according to predicates. This is in the same spirit as the XQFT data model described in [28]. The formal semantics of XFT is shown in Figure 2 and detailed next. The selection operators in XFT are defined for the binding and the existential predicate semantics. R_i denote matching tables.

- **get**(k) returns a table containing one tuple t for each node n with a non-empty set of matches (denoted $matches(n, k)$) of term k against the subtree rooted at n. In Figure 2, $\mathcal{N}$ denotes the set containing a unique identifier for each node in the input document collection. The result of evaluating **get**(`services`) on the document on Figure 1 is a table containing one entry for each of `congress-info`, `bill`, `legis-body`, `legis`, `legis-session` and `legis-desc`.
- R_1 **or** R_2 returns a table which collects for each node n and pattern p, the union of the corresponding matches found in R_1 and in R_2.
- The conjunction operator R_1**and**R_2 creates a new table for the nodes with matches given by both R_1 and R_2. For each such node n, if R_i states that the matches of pattern p_i under n are the set m_i, we may infer that n actually contains matches of pattern $p_1 \circ p_2$. The operator $\circ$ concatenates two patterns, eliminating duplicate terms. For instance, $(k_1, k_2) \circ (k_3, k_2, k_4) = (k_1, k_2, k_3, k_4)$. The empty pattern $()$ is the identity element: $p \circ () = () \circ p = p$. All elements contain a match of $()$. It is easy to see that the matches of $p_1 \circ p_2$ are given by the natural join of $m_1 \bowtie m_2$.

Since the expressions delivering the operands of **and** may contain **or**, the join of the corresponding matching tables can in principle construct several distinct tuples agreeing on the N and P attributes. In this case, **group** coalesces all matches by unioning them together.

- R_1**minus**R_2 returns the tuples in R_1 pertaining to nodes without matches in R_2.

Summarizing, operators **get**, **and**, **or** and **minus** find for each inclusion-exclusion pattern pair *pp* specified by the expression, the nodes n with no matches of $pp.E$ and the actual matches of $pp.I$ under n.

The remaining operators, also referred to as full-text predicates, test various conditions on the matches (possibly filtering them in the process). For each full-text predicate P, we provide two operators P^b and $P^\exists$, to support binding, respectively existential semantics. In Figure 2, K denotes an ordered pattern of terms $(k_1, \ldots, k_l)$, and $\Pi_K(M)$ the projection of the set of matches M on the components corresponding to the terms in K. If the matches in M do not correspond to a pattern which includes the terms in K, $\Pi_K(M) = \emptyset$.

- $\textsf{\textbf{times}}_{\theta c}(\{K_1, \ldots, K_l\})$ applied to an input table R, selects the tuples whose cumulative match count for patterns $K_1, \ldots, K_l$, satisfies the θ-comparison to the integer c.
- $\textsf{\textbf{ordered}}^\exists(k_1, \ldots, k_l)$ applied to table R returns the tuples containing some match in which the position of term k_i appears in the document order before that of the term k_{i+1} for all i in the input term list $k_1, \ldots, k_l$. The other flavor of this operator, $\textsf{\textbf{ordered}}^b(k_1, \ldots, k_l)$, drops from each tuple t the matches in $t.M$ which violate the above ordering condition. If all matches are dropped, so is t.
- $\textsf{\textbf{window}}^\exists_{\theta c}(k_1, \ldots, k_l)$ applied to an input table R, returns the tuples $t \in R$ containing some match which fulfills the window condition, i.e. in which all positions of terms $k_1, \ldots, k_l$, lie within a maximum distance which satisfies the θ-comparison with integer c. $\textsf{\textbf{window}}^b_{\theta c}(k_1, \ldots, k_l)$ drops all non-conforming matches first (dropping also t if no matches are left).
- $\textsf{\textbf{dist}}^\exists_{\theta c}(k_1, \ldots, k_l)$ applied to an input table R, returns the tuples $t \in R$ containing some match which conforms to the condition that the positions of terms which are adjacent in the pattern $(k_1, \ldots, k_l)$, are at distance satisfying the θ-comparison with integer c. $\textsf{\textbf{dist}}^b_{\theta c}(k1, \ldots, k_l)$ drops all non-conforming matches first (and also t if no matches are left).

Example 2.2. *The query in Example 1.1 is expressible in* XFT *as (we abbreviate the terms for readability):*

$$\sigma_{\textsf{\textbf{ordered}}^\exists(\texttt{Jeff},\texttt{edu})}(\sigma_{\textsf{\textbf{window}}^\exists_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textsf{\textbf{get}}(\texttt{Jeff})\ \textsf{\textbf{and}}\ \textsf{\textbf{get}}(\texttt{edu}))).$$

Here we used **ordered**$^\exists$ *and* **window**$^\exists$ *to denote the existential semantics where an answer must contain at least one match of* `Jefferson` *and* `education` *that satisfies window and one possibly different match that satisfies order.* [2]

We now formalize the connection between the results of XFT algebra expressions and the matches of patterns against the input XML trees. Like all XQFT-class queries, each XFT expression X specifies a set of inclusion-exclusion pattern pairs denoted with *ppairs*(X) and defined formally in Figure 3. The XFT operators omitted from the figure do not affect the set of pattern pairs. The inclusion-exclusion pattern pair of the expression in Example 2.2 is (I = (`Jefferson`, `education`), $E = \{\}$). The query contains no negation and therefore specifies no exclusion pattern.

Operator	Definition
$\textsf{\textbf{get}}(k)$	$\{t \mid n \in \mathcal{N}, M = \mathit{matches}(n,k), M \neq \emptyset,$ $t.N = n, t.P = (k), t.M = M\}$
$R_1\textsf{\textbf{or}}R_2$	$\textsf{\textbf{group}}(R_1 \cup R_2)$
$R_1\textsf{\textbf{and}}R_2$	$\textsf{\textbf{group}}(\{t \mid t_1 \in R_1,\ t_2 \in R_2,\ t.N = t_1.N = t_2.N,$ $t.P = t_1.P \circ t_2.P, t.M = t_1.M \bowtie t_2.M\})$
$R_1\ \textsf{\textbf{minus}}R_2$	$\{t \mid t \in R_1, t.N \not\in \Pi_N(R_2)\}$
$\sigma_{\textsf{\textbf{times}}_{\theta c}(S)}(R)$	$\{t \mid t \in R, (\sum \bigcup_{K \in S}\{\lvert\Pi_K(t_1.M)\rvert \mid t_1 \in R,$ $t_1.N = t.N\})\ \theta\ c\}$
$\sigma_{\textsf{\textbf{ordered}}^\exists(K)}(R)$	$\{t \mid t \in R,\ \exists(m_1, m_2, \ldots, m_l) \in \Pi_K(t.M),$ $m_1 < m_2 < \ldots m_l\}$
$\sigma_{\textsf{\textbf{ordered}}^b(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P,$ $M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t_1.M),$ $m_1 < m_2 < \ldots m_l\},$ $M \neq \emptyset, t.M = M\}$
$\sigma_{\textsf{\textbf{window}}^\exists_{\theta c}(K)}(R)$	$\{t \mid t \in R,\ \exists(m_1, m_2, \ldots, m_l) \in \Pi_K(t.M),$ $(\max_{1 \leq i,j \leq l} \mathit{distance}(m_i, m_j))\ \theta\ c\ \}$
$\sigma_{\textsf{\textbf{window}}^b_{\theta c}(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P,$ $M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M),$ $(\max_{1 \leq i,j \leq l} \mathit{distance}(m_i, m_j))\theta\ c\},$ $M \neq \emptyset, t.M = M\}$
$\sigma_{\textsf{\textbf{dist}}^\exists_{\theta c}(K)}(R)$	$\{t \mid t \in R,\ (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M),$ $\mathit{Order}(m), \bigwedge_{1 \leq i < l} \mathit{distance}(m_i, m_{i+1})\ \theta\ c\}$
$\sigma_{\textsf{\textbf{dist}}^b_{\theta c}(K)}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P,$ $M = \{m \mid m = (m_1, m_2, \ldots, m_l) \in \Pi_K(t.M),$ $\mathit{Order}(m), \bigwedge_{1 \leq i < l} \mathit{distance}(m_i, m_{i+1})\ \theta\ c\},$ $M \neq \emptyset, t.M = M\}$
$\textsf{\textbf{group}}(R)$	$\{t \mid t_1 \in R, t.N = t_1.N, t.P = t_1.P,$ $t.M = \bigcup\{t_2.M \mid t_2 \in R, t_2.N = t_1.N,$ $t_2.P = t_1.P\},$ $t.M \neq \emptyset\}$
θ	$\in \{=, <, \leq, >, \geq\}$

Figure 2: The XFT **Algebra**

Theorem 2.1. *Let X be an* XFT *algebra expression consisting only of* **get**, **and**, **or** *and* **minus** *operators. Then the result of X on any collection of XML documents is a matching table R such that $t \in R$ if and only if there is some pattern pair $pp \in ppairs(X)$ where*
(i) $t.P = pp.I$,
(ii) $t.M$ holds all matches of $pp.I$ under $t.N$, and
(iii) there are no matches of $pp.E$'s patterns under $t.N$.

2.2 XFT Rewriting Optimization

The main benefit of using an algebra is to be able to apply logical rewritings to queries while preserving the set of query answers. The design of XFT is highly influenced by the relational algebra in order to enable typical rewritings such as pushing selections and join reordering. While a full-fledged rewriting-based optimizer is subject of future work,

$$\begin{aligned}
\mathit{ppairs}(\textsf{\textbf{get}}(k)) &= \{pp \mid pp.I = (k), pp.E = \{\}\} \\
\mathit{ppairs}(X_1\textsf{\textbf{and}}X_2) &= \{pp \mid pp_1 \in \mathit{ppairs}(X_1),\ pp_2 \in \mathit{ppairs}(X_2), \\
&\qquad pp = pp_1 \bullet pp_2\} \\
\mathit{ppairs}(X_1\textsf{\textbf{or}}X_2) &= \mathit{ppairs}(X_1) \cup \mathit{ppairs}(X_2) \\
\mathit{ppairs}(\neg X) &= \{(x_1 \bullet \cdots \bullet x_n) | \{pp_1, \ldots, pp_n\} = \mathit{ppairs}(X), \\
&\qquad x_1 \in \mathit{not}(pp_1), \ldots x_n \in \mathit{not}(pp_n)\}
\end{aligned}$$

$$(I_1, E_1) \bullet (I_2, E_2) = (I_1 \circ I_2, E_1 \cup E_2)$$
$$\mathit{not}((I, E)) = \{((), \{I\})\} \cup \{(e, \{\}) | e \in E\}$$

Figure 3: Inclusion-Exclusion pairs of patterns in XFT **expressions**

[2] To relax the query further and return answers containing only one term, we could replace **and** by an outer join.

our experiments already confirm the expected performance benefit of rewriting-based optimization (Section 4).

The formal semantics of XFT implies a plethora of algebraic equivalences. We only list a few here that we implemented as a proof of concept. Let us consider again the query in Example 2.2. We have seen that its expression in the algebra is (we abbreviate the terms for readability):
$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff,edu})}(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff,edu})}(\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu}))).$
This expression is equivalent to
$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff,edu})\ \wedge\ \textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff,edu})}(\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu}))$
and to
$\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff,edu})}(\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff,edu})}(\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu}))).$

Now consider adding to this query the condition that the term `services` appear after `education` in the text. This query could be written as

$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff,edu})}(\sigma_{\textbf{ordered}^{\exists}(\texttt{edu,services})}(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff,edu})}($
$\textbf{get}(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu})\ \textbf{and}\ \textbf{get}(\texttt{services})))),$

which is equivalent after pushing selections into the **and** operator, to
$\sigma_{\textbf{ordered}^{\exists}(\texttt{Jeff,edu})}(\sigma_{\textbf{window}^{\exists}_{\leq 10}(\texttt{Jeff,edu})}(get(\texttt{Jeff})\ \textbf{and}\ \textbf{get}(\texttt{edu}))$
$\textbf{and}\ (\sigma_{\textbf{ordered}^{\exists}(\texttt{edu,services})}(\textbf{get}(\texttt{edu})\ \textbf{and}\ \textbf{get}(\texttt{services})))).$

Finding LCAs of query terms is a typical implementation of the **and** operator which has been widely used in previous work [17, 21, 24, 29]. However, as we have shown in Example 1.1, simply computing LCAs and applying selection predicates does not always return the set of correct answers due to violating the above equivalences.

Section 3 shows our solution to combining the best of two worlds in algorithms that preserve query rewritings and enable the implementation of the **and** operator using LCAs.

2.3 Scoring

The goal of scoring in the context of XFT is twofold: (i) allow to manipulate scored answers in a query plan while preserving the query semantics and, (ii) define answer scores in a way that guarantees that the score of an element node can be computed efficiently. We describe a scoring method that conforms to these requirements.

We assume that scores are stored along with tuples in the initial matching tables and define term weights as follows:

Term Weights. We adapt the standard *tf*idf* function [16, 10] to individual nodes and compute the weight of a term k for a given leaf node n. This function is defined as: (i) *idf*, or inverse document frequency, that quantifies the relative importance of an individual term in the collection of documents; and (ii) *tf*, or term frequency, that quantifies the relative importance of a term in an individual document. In the vector space model [3], query terms are assumed to be independent of each other, and the *tf*idf* contribution of each term is added to compute the final score of the answer document. Intuitively, since XML queries return elements as opposed to whole documents, the weight of a term in elements of different types (tag) may be different. We denote *itf*, the *idf* of elements of the same type.

The term frequency *tf(n,k)* is defined as ($occ(k, n)$ denotes the number of distinct occurrences of term k in leaf node n):

$$tf(n,k) = \frac{occ(k,n)}{max\{occ(k',n) \mid k' \in words(n)\}}$$

Let T be the set of all nodes that share the same tag as node n, then, *itf(n,k)* is defined as:

$$itf(n,k) = \log(1 + \frac{\mid T \mid}{\mid \{n \in T \mid k \in words(n)\} \mid})$$

Intuitively, `bill` elements have a different relevance to a given term than `committee-info` elements. The fewer elements of the same type, the higher their *itf* for a term.

We note that the *tf* of a node for a term can be inferred from its descendant nodes while *itf* needs to be pre-computed and stored with the node. The **get** operator in our algebra could be used to retrieve term weights. Next, we define an element score.

Answer Scores. We define the weight of a term k in an answer s as $\texttt{tf}(\texttt{s},\texttt{k}) \times \texttt{itf}(\texttt{s},\texttt{k})$. Given a pattern containing a set of keywords $\mathcal{K}$, the score of an answer is defined as: $\texttt{score}(\texttt{s}) = \sum_{\texttt{k}\in\texttt{K}} (\texttt{tf}(\texttt{s},\texttt{k}) \times \texttt{itf}(\texttt{s},\texttt{k}))$.

Given this definition, it is easy to see that **and** can compute the score of each output tuple using the above formula while **or** and **minus** would only need to preserve input scores. Binding predicates may choose to modify scores in a way that is different from existential ones thereby enforcing query semantics when computing scores. Consequently, the use of an algebra permits better control over the intermediate answer scores and decide whether or not individual query operators have an impact on scores.

Similarly to vector-based scoring, our method assumes independence between term weights within an element node. Thus, a *key* advantage of our scoring method is the ability to compute the score of a node in an incremental fashion from its descendant nodes without affecting the algorithms complexity. More sophisticated scoring is possible (though with higher evaluation complexity) if the independence assumption is relaxed, as in probabilistic IR models [15]. We do not pursue that avenue further in this paper.

2.4 Application to XQFT and NEXI

XQFT. XQuery Full-Text (XQFT), an upcoming W3C standard [28], is an extension of XPath and XQuery to allow with full-text predicates. Wherever XQuery allows a predicate, XQFT allows the expression **ftcontains**(E) with E a text search expression. For example,

/*books*/*book*[*review* **ftcontains**

((*“thumbs”*&& *“up”* || *“must”*&& *“read”*) **distance** ≤ 1

||(*“best-seller”* **times** ≥ 2))]/*author*

returns all authors of books whose very enthusiastic review contains either the term pair (“thumbs”,“up”) in close proximity or similarly the pair (“must”,“read”) , or mentions the term “best-seller” at least twice.

The syntax of the XQFT expressions that may appear in the scope of the **ftcontains** expression is given in Figure 4. The XQFT primitives not shown in the figure do not affect the patterns as they do not mention any terms. We define the binding semantics of XQFT in Figure 5.

NEXI. Similarly to XQFT, we show how our algebra captures the semantics of NEXI [26], the language that is being used within INEX [19] to express XML search queries. The core expression in NEXI is the **about** expression which permits conjunction of query terms. For example,

/*books*/*book*[**about**(*review*, *“thumbs” “up” “must” “read”*

“best-seller”)]/*author*

$$
\begin{array}{lcl}
E & \rightarrow & {}''k'' \\
 & | & E_1 \;\&\&\; E_2 \\
 & | & E_1 \;||\; E_2 \\
 & | & E \textsf{ times } \theta \; c \\
 & | & E \textsf{ ordered} \\
 & | & E \textsf{ window } \theta \; c \\
 & | & E \textsf{ distance } \theta \; c \\
 & | & (E) \\
 & | & \neg E \\
\theta & \rightarrow & = \; | \; < \; | \; \leq \; | \; > \; | \; \geq \\
k & \rightarrow & \textit{any term}
\end{array}
$$

Figure 4: Syntax of W3C's Standard XQFT

$$
\begin{array}{rcl}
\llbracket {}''k'' \rrbracket & = & \textsf{get}(k) \\
\llbracket E_1 \;\&\&\; E_2 \rrbracket & = & \llbracket E_1 \rrbracket \textsf{ and } \llbracket E_2 \rrbracket \\
\llbracket E_1 \;||\; E_2 \rrbracket & = & \llbracket E_1 \rrbracket \textsf{ or } \llbracket E_2 \rrbracket \\
\llbracket E \textsf{ predicate} \rrbracket & = & \textsf{or}_{K_i \in S}(\sigma_{\textsf{predicate}(K_i)}(\llbracket E \rrbracket)), \text{ where} \\
 & & \textsf{predicate} \in \{\textsf{ordered}^b, \textsf{window}^b_{\theta c}, \textsf{dist}^b_{\theta c}\} \\
 & & \text{and } S = \Pi_I(ppairs(E)) \\
\llbracket E \textsf{ times } \theta \; c \rrbracket & = & \sigma_{\textsf{times}_{\theta c}(S)}(\llbracket E \rrbracket), \qquad\qquad \text{(see } S \text{ above)} \\
\llbracket (E) \rrbracket & = & \llbracket E \rrbracket \\
\llbracket \neg E \rrbracket & = & \{t \mid n \in \mathcal{N}, t.N = n, t.P = (), t.M = \{()\}\} \\
 & & \textsf{minus } \llbracket E \rrbracket
\end{array}
$$

Figure 5: Specification of XQFT Semantics in XFT

returns book authors whose review *is about* one of the terms. NEXI also allows the specification of weights which is not yet supported in XFT. The syntax of NEXI **about** is very simple: $E \rightarrow {}''k''|E_1\ E_2$. We express its semantics through translation into XFT: $\llbracket {}''k'' \rrbracket = \textsf{get}(k)$, $\llbracket E_1\ E_2 \rrbracket = \llbracket E_1 \rrbracket$ **or** $\llbracket E_2 \rrbracket$.

3. XFT EVALUATION ALGORITHMS

Algorithms to compute LCAs of query terms have shown very good performance for the evaluation of conjunctive keyword queries [17, 21, 24, 29]. We have seen in Section 1 that their applicability to the evaluation of full-text predicates is not straightforward. This section presents novel algorithms that implement operators in the algebra proposed in Section 2. Section 3.1 describes AllNodes a straightforward implementation of our algebra. More efficient algorithms that use element nesting, are presented in Section 3.2. We finish with a note on incremental scoring in SCU .

3.1 The AllNodes Algorithm

A key design goal of the XFT algebra was amenability to efficient, set-at-a-time pattern match construction and filtering by leveraging tried-and-true relational techniques. Indeed, notice in Figure 2 that R_1**and**R_2 joins R_1 and R_2 on N, aggregating the M attributes of joining tuples via natural join; R_1**or**R_2 corresponds to taking the union of R_1 and R_2, grouping it by the N and P attributes, and aggregating the M attributes by unioning them; both flavors of **ordered**, **window** and **distance** are simple selections over the M attribute of each tuple, and **times** can be evaluated by grouping on N followed by aggregation using the count function, with a `having` clause for the θ-comparison.

We have implemented this evaluation strategy as a proof of concept in the AllNodes algorithm detailed next.

Node and Word Identifiers. We choose to represent node and term match identifiers using the well-established Dewey encoding which enables efficient computation of the LCA of two nodes (or term matches) as the longest common prefix of their ids [17, 21, 24, 29]. For example, the id of the first match of the term `services` encountered in the document in Figure 1(b) is 1.1.1.9 where 1 is the id of the root node `bill`, 1.1 is the id of node `congress-info` and 1.1.1 is the id of the `text` node containing the term. Also, testing whether node a is an ancestor of node d reduces to finding a's id as a prefix of d's id.

The get operator. The **get** operator is implemented using a lookup in a standard inverted list index [3] IL which associates with each term k the list of its matches, given by their Dewey ids. The list of matches is stored in top-down, right-child-first traversal order. **get**(k) needs to return all nodes containing the match of k, but the inverted lists only store *the immediately containing node*, i.e., leaf nodes in the XML document. While this requires some processing when reading the inverted lists, the alternative of storing all ancestor nodes of a term match is known to lead to tremendous space overhead and is commonly avoided [17]. The processing required to restore all ancestors of a term match is minimal: we obtain their ids as the strict prefixes of the id of the match. The implementation of **get** must

1. perform a single pass over the input inverted lists.
2. collect for each node all term matches under it before outputting the node;
3. avoid duplicate output of node ids.

We satisfy all these desiderata using Algorithm 1 below.

Algorithm 1 get(k) **in AllNodes Implementation**

```
Require: inverted list $IL$ stores matches of $k$ in top-down,
         rightmost-child-first traversal order
Ensure: outputs matching table sorted in descending lexico-
        graphic order on $N$
initialize stack
push($t$), where $t.N$ = the root, $t.P = (k)$, $t.M = \emptyset$
$(m, n)$ = get_next_match($IL, k$)
{ $m$ is the match of $k$, $n$ its immediately containing node}
while (stack not empty) do
    t=top(stack)
    if $n$ is a descendant-or-self of $t.N$ then
        add $m$ to all tuples in stack
        for (each proper descendant $d$ of $t.N$
                which is an ancestor-or-self of $n$) do
            push($t_d$), where $t_d.N = d, t_d.P = (k), t_d.M = \{m\}$
            {push higher nodes first!}
        end for
        (n,m) = get_next_match($IL, k$)
    else
        output $t$ and pop the stack
    end if
end while
```

The algorithm produces the output sorted by the descending lexicographic order of the Dewey ids of the N attribute. Thus, the task of subsequent operators is to preserve this order to enable merge-style algorithms, as detailed below.

The and operator. The **and** implementation performs a merge of the two (already sorted) inputs. Whenever tuples

t_1 and t_2 have the same value for their N attribute, the set of matches of the resulting tuple t is computed by taking the natural join of $t_1.M$ with $t_2.M$. This is in keeping with Theorem 2.1, since all matches of the combined pattern under $t.N$ are indeed found this way. The results of the joins can be large, degenerating in most cases (when the patterns in t_1 and t_2 do not overlap) to Cartesian products. These are notoriously expensive to compute, both in terms of time and space. This computation can be performed lazily, sometimes avoided entirely, and in most cases preceded by a reduction of the operand sizes. The idea is to record the two operands of the natural join (worst-case Cartesian product) in $t.M$ without further computation. To this end, $t_1.M, t_2.M, t.M$ hold *lists* of sets of matches, with the understanding that a list represents the multiway join of all its sets, in the order they are listed. We describe below how these matches are materialized when evaluating full-text predicates.

The or operator. **or** essentially unions its operands R_1 and R_2, grouping the result by the N and P attributes and unioning together the matches in each group. The union is computed by simply merging the operands, which are already sorted on N, breaking ties by sorting on P. The merge ensures that the output is ordered on N, P as well. The patterns are compared using lexicographic order induced by the alphabetic order of the individual terms. The grouping involves no additional overhead, as the operands are already sorted on the grouping attributes so each group is listed contiguously in the merge of R_1 and R_2.

The times operator. **times** needs to group its operand table on N, counting the matches in each group. This requires a linear scan, since the table is already sorted on N.

The Other Operators. All other operators are order-preserving, as they at most drop tuples from the table. They require one linear scan of the input table.

Avoiding Cartesian product. Recall that the **and** operator does not materialize matches, instead simply computing the list of $t.M$ as the append of the list $t_2.M$ and the list $t_1.M$. As long as the result of this **and** operator is consumed by other **and** operators, matches won't need to be materialized and the lists keep growing. Materialization is delayed until required by a predicate, at which point the Cartesian product is pruned by pushing selections into it, using equivalences in the spirit of those illustrated in Section 2.2.

Pipelining. All our operator implementations can be pipelined (the **group** operator used to implement **and** and **or** is not fully blocking, as its input is sorted).

3.2 The SCU Algorithm

The definition of matching tables was primarily introduced with the purpose of providing a clean formal semantics of XML full-text search which captures various query language semantics, in particular XQFT (as shown in Section 2.4). Matching tables also enable relational-style evaluation of full-text queries, as demonstrated in Section 3.1.

However, matching tables are not necessarily the ideal data structure to manipulate during evaluation. Indeed, by definition of the matching table, whenever a table R contains a tuple t, R also contains a tuple t_a for each ancestor a of $t.N$, and $t.M$ is included in $t_a.M$. $t.M$ is therefore redundantly stored in R, forcing the AllNodes algorithm (especially the full-text predicates) to repeatedly visit its contents. The redundancy increases with element nesting, i.e., the depth of node t_d in the document.

In this section, we introduce an alternative evaluation algorithm that operates on tables which eliminate precisely the redundant storage of the descendant's matches in the ancestor. These tables, called SCU tables (for Smallest Containing Unit), lead to smaller intermediate results and therefore to potentially better overall performance. Our experiments (Section 4) show that this potential is indeed reached.

Definition 3.1 (SCU **tables**). SCU *tables have the same schema as matching tables, but satisfy:*

1. *for every pair of tuples t_a, t_d such that $t_a.N$ is an ancestor of $t_d.N$ and $t_a.P = t_d.P$, the descendant's matches are not repeated in the ancestor's: $t_a.M \cap t_d.M = \emptyset$. We call $t_a.M$ the* direct *matches of node $t_a.N$, and $t_d.M$ the* inherited *matches of node $t_a.N$.*
2. *there is no tuple t_a such that $t_a.M = \emptyset$ and such that $|\{t_d \mid t_d.P = t_a.P, t_d \text{ is descendant of } t_a\}| = 1$.*

An immediate implication on SCU tables is that in any tuple t, $t.N$ is the LCA of all matches in $t.M$.

Example 3.1. *Recall the matching table of the term* `education` *in Example 2.1. Its corresponding* SCU *table is given below, this time revealing the Dewey ids of the nodes and positions:*

N	P	M
1.1	$\langle$`education`$\rangle$	$\{(1.1.1.3)\}$
1.2.2.2	$\langle$`education`$\rangle$	$\{(1.2.2.2.1.45)\}$
1.3.2	$\langle$`education`$\rangle$	$\{(1.3.2.1.67)\}$

SCU tables contain the same information as matching tables, but in a compressed way. Any matching table can be reduced into an SCU table: for each tuple t_a, remove from $t_a.M$ the set $t_d.M$ whenever $t_d.N$ is a descendant of $t_a.N$ and $t_a.P = t_d.P$, and drop t_a altogether if $t_a.M$ becomes empty and if there is precisely one tuple t_d with $t_d.P = t_a.P$ and where $t_d.N$ is a descendant of $t_a.N$. We call this operation `reduce`. Its inverse is called `expand` and it turns an SCU table into a matching table in the obvious way.

The SCU algorithm takes an XFT expression E and a list of SCU tables $S_1, \ldots, S_n$ and returns an SCU table $SCU_E(S_1, \ldots, S_n)$. The SCU algorithm is semantically equivalent to the XFT algebra, in the sense that for any matching tables $R_1, \ldots, R_n$ and any XFT expression E,

$$\begin{gathered} E(R_1, \ldots, R_n) \\ = \\ \texttt{expand}(SCU_E(\texttt{reduce}(R_1), \ldots, \texttt{reduce}(R_n))). \end{gathered} \qquad (\dagger)$$

SCU tables are possible due to the XML tree structure since they rely on element nesting information stored in Dewey ids to expand into matching tables. SCU tables lead to a significant simplification in the implementation of the **get** operator, which now needs to output for each term match only the immediately containing node but none of the ancestors. The implementation of the **ordered**^{b}, **dist**^{b}, **window**^{b} operators is not affected, as all they do is filter matches, dropping the ones who violate the filtering condition regardless of whether they are stored redundantly. There is no change of semantics if these full-text predicates work on SCU tables instead of matching tables. However, the runtime performance is improved as fewer tuples are inspected, and shorter lists of matches per tuple are scanned.

The conciseness of SCU tables comes at a price though: it poses new problems to the evaluation of the other full-text predicates, as well as the **and** operator, in the sense that directly applying their AllNodes implementation to SCU tables breaks requirement (†), as detailed below.

and can no longer be computed as an equijoin on the N attribute, as illustrated by the following example.

Example 3.2. *Consider the* XFT *expression*

$$\textbf{get}(\texttt{Column})\ \textbf{and}\ \textbf{get}(\texttt{introduced})$$

For the document in Figure 1(a), there is only one match of the term `Column` *and of* `introduced`*. The immediately containing nodes are* `sponsors`*, respectively* `action-desc`*. Let* $R_{\texttt{Column}}$ *and* $R_{\texttt{introduced}}$ *be the* SCU *tables returned by* **get**(`Column`) *and* **get**(`introduced`) *according to the* SCU *implementation. These contain one tuple each,* $t_{\texttt{Column}} \in R_{\texttt{Column}}$ *with* $t_{\texttt{Column}}.N = \texttt{sponsors}$*, and similarly for* $t_{\texttt{introduced}} \in R_{\texttt{introduced}}$*. The LCA* `bill` *of* `sponsors` *and* `action-desc` *contains a match of pattern* (`Column`, `introduced`)*, but the equijoin of* $t_{\texttt{Column}}$ *with* $t_{\texttt{introduced}}$ *is empty, failing to produce the corresponding tuple. The* `expand` *operation cannot remedy the problem, as the expansion of an empty table remains empty. In contrast, by Theorem 2.1, since* (`Column`, `introduced`) *has a match under* `bill`*,* **and** *should output such a tuple as is indeed the case in the* AllNodes *implementation.*

The **ordered**$^\exists$,**dist**$^\exists$, **window**$^\exists$ and **times** operators are affected as well if operating on SCU tables. The semantics of full-text predicates depends on the *all* matches appearing under a node, which is why in the matching table, each tuple t is *self-contained* for the purposes of predicate evaluation: all matches under node $t.N$ are collected in $t.M$. The full-text predicates can therefore be evaluated locally on each tuple, *akin to predicates in relational selections*. In contrast, an SCU table keeps the matches relevant to the evaluation of a full-text predicate on node $t_a.N$ *distributed across several tuples* corresponding to descendants of $t_a.N$. In this case, full-text search predicates are turned into global aggregation operations working on the entire table. This is illustrated in Example 1.1. We summarize it here.

Example 3.3. *Recall that the query in Example 1.1 is* $\sigma_{\textbf{ordered}^\exists(\texttt{Jeff},\texttt{edu})}(\sigma_{\textbf{window}^\exists_{\leq 10}(\texttt{Jeff},\texttt{edu})}(\textbf{get}(\texttt{Jeff})\textbf{and}\ \textbf{get}(\texttt{edu})))$ *Node* `action` *is returned since it contains one match for each of the query terms satisfying* **ordered**$^\exists$ *and one satisfying* **window**$^\exists$*. However, since the match satisfying* **ordered**$^\exists$ *(resp.,* **window**$^\exists$*) violates* **window**$^\exists$ *(resp.,* **ordered**$^\exists$*), each match would be filtered prematurely regardless of the order of predicate application.*

We compensate for these problems by adapting the implementation of the affected operators as follows.

The and operator. The previous example shows that the equijoin does not work on SCU tables, and that two tuples $t_1 \in R_1, t_2 \in R_2$, despite not agreeing on their N attribute, carry matches relevant to the LCA of $t_1.N$ and $t_2.N$. This suggests an immediate fix: when "joining" tuples t_1 and t_2, output the tuple t with $t.N = LCA\ (t_1.N, t_2.N), t.P = t_1.P \circ t_2.P, t.M = t_1.M \bowtie t_2.M$.

This approach poses significant efficiency challenges. Since any two nodes from the same document have an LCA (the document root in the worst case), the tempting but naive implementation involves a Cartesian product. A more efficient alternative would be to adopt one of the state-of-the-art stack-based algorithms developed in prior work to evaluate conjunctive keyword searches in XML documents by computing the LCAs of the matches, without keeping track of the matches themselves [29], or without evaluating any full-text predicates on them [4, 17].

Preorder versus Postorder Incompatibility. The immediate adaptation of existing stack-based algorithms to our needs is precluded by the fact that existing algorithms assume the input sorted in *preorder*, but produce their result in *postorder*. This is not a problem in prior works, which focus only on computing LCAs of a conjunctive patterns without post-processing the result. In our setting this mismatch precludes operator compositionality.

Compositional Stack-Based Algorithms. We propose a stack-based solution yielding an efficient single-pass algorithm which computes S_1 **and** S_2, where S_1, S_2 are SCU tables sorted in postorder traversal order. The result is also sorted in postorder, thus facilitating the seamless composition of **and** operators with each other (and, as we shall see below, with all other operators) without any intervening sorting step. This is important for evaluation performance, and essential for enabling pipelined implementation. Though stack-based processing is not a new idea, the solution for consuming input in postorder is novel and guarantees efficient compositionality of our algebra operators.

Algorithm 2 uses a stack containing descendant LCAs and their matches for consideration by ancestor LCAs.

The stack holds tuples of schema (L, Dir_1, Dir_2) and the algorithm maintains the invariant that, according to the input consumed so far,

- L is an LCA of at least one pair of matches from $S_1 \times S_2$;
- Dir_i is a set of matches (called the *direct matches*) from S_i, each contained in L but occurring in no LCA which is a proper descendant of L;

The stack is maintained while two cursors are advanced in a single pass over the input SCU tables S_1 and S_2, reading tuples s_1, s_2 respectively. At each step, the stack contains a list of LCAs of pairs of matches from S_1, S_2. These LCAs reside on the same root-to-leaf path, with the deepest LCA at the top of the stack. This is achieved by pushing a newly computed LCA only if it is a descendant of the stack top's LCA, $top().L$.

If the new LCA l is greater in postorder than $top().L$ (line 22 in Algorithm 2), the postorder sorting of the inputs guarantees that no further descendant LCAs of $top().L$ can be encountered and we can pop and output the latter (lines 26–27). Additionally, if the newly computed LCA l is not an ancestor of $top().L$, there is a new LCA l' induced by $top().L$ and l (computed at line 24), and l' must be pushed (line 41) on the stack before l (line 44), to maintain the descendant-last invariant for stack LCAs. Notice that l' has no direct matches, since they are all nested within l and $top().L$.

If the new input contributes to the same LCA as $top().L$ (line 30), this input is recorded in the top stack tuple (lines 30–36), since we need to accumulate all direct matches contributing to the LCAs.

Finally, the new input can generate an LCA l which is a descendant of $top().L$. At this point (line 36), we know that the input matches are not direct matches for $top().L$, and they must be removed from the $top().Dir_i$ lists (lines 37–40). Moreover, l is pushed on the stack since now we expect the remaining input to contribute to its descendant LCAs (line 44).

When a stack tuple o is output, we generate SCU table tuples t from it. This involves setting $t.N = o.L$ and computing the matches corresponding to the pairs in the Cartesian product $o.Dir_1 \times o.Dir_2$ (not shown in the pseudocode).

The node operations performed by Algorithm 2 are all very well supported by Dewey ids. Indeed, checking that l is larger than $top().L$ in postorder (line 22) reduces to checking that the Dewey id of l is either lexicographically larger than or a strict prefix of the Dewey id of $top().L$. Similarly for the tests in line 14. The ancestor test in line 26 reduces to testing that the Dewey id of l' is a strict prefix of $top().L$'s id. The LCA computations are implemented as simply finding the longest common prefix of the operands.

Algorithm 2 SCU Implementation of **and** Operator

```
Require: $S_1, S_2$ are SCU tables sorted in postorder on $N$
Ensure: outputs SCU table corresponding to
      reduce(expand($S_1$) and expand($S_2$))
      sorted in postorder on $N$ attribute
 1: initialize stack
 2: open cursors on $S_1$ and on $S_2$
 3: $s_1 \leftarrow$ get_next($S_1$), $s_2 \leftarrow$ get_next($S_2$)
 4: $l \leftarrow LCA\ (s_1.N, s_2.N)$
 5: push($L = l, Dir_1 = \{s_1\}, Dir_2 = \{s_2\}$)
 6: while (at least one cursor can advance) do
 7:    if ($EOF(S_2)$) then
 8:       $s_1 \leftarrow$ get_next($S_1$);
 9:    else if ($EOF(S_1)$) then
10:       $s_2 \leftarrow$ get_next($S_2$)
11:    else
12:       $s'_1 \leftarrow$ look_ahead($S_1$); $l_1 \leftarrow LCA\ (s'_1.N, s_2.N)$
13:       $s'_2 \leftarrow$ look_ahead($S_2$); $l_2 \leftarrow LCA\ (s_1.N, s'_2.N)$
14:       if (($l_1 <_{post} l_2$ or
               $l_1 = l_2$ and $s_1.N \leq_{post} s_2.N$) then
15:          $s_1 \leftarrow$ get_next($S_1$)
16:       else
17:          $s_2 \leftarrow$ get_next($S_2$)
18:       end if
19:    end if
20:    $l \leftarrow LCA\ (s_1.N, s_2.N)$
21:    $l' \leftarrow l$
22:    if ($l >_{post} top().L$) then
23:       if ($l$ is not ancestor of $top().L$) then
24:          $l' \leftarrow LCA\ (l, top().L)$
25:       end if
26:       while ($l'$ is ancestor of $top().L$) do
27:          $o \leftarrow$ pop(); output $o$
28:       end while
29:    end if
30:    if ($l' = top().L$) then
31:       if ($S_1$ cursor was last to advance) then
32:          $top().Dir_1 \leftarrow top().Dir_1 \cup \{s_1\}$
33:       else
34:          $top().Dir_2 \leftarrow top().Dir_2 \cup \{s_2\}$
35:       end if
36:    else
37:       if (non-empty stack) then
38:          $top().Dir_1 \leftarrow top().Dir_1 \setminus \{s_1\}$
39:          $top().Dir_2 \leftarrow top().Dir_2 \setminus \{s_2\}$
40:       end if
41:       if ($l' \neq l$) then
42:          push($L = l', Dir_1 = \emptyset, Dir_2 = \emptyset$)
43:       end if
44:       push($L = l, Dir_1 = \{s_1\}, Dir_2 = \{s_2\}$)
45:    end if
46: end while
47: while (non-empty stack) do
48:    $o \leftarrow$ pop(); output $o$
49: end while
```

Evaluation of Full-text Predicates. Since each SCU tuple contains only the direct matches under its nodes, but the predicates depend also on indirect matches, their evaluation needs to fulfill the following requirements:

- detect that a descendant already satisfied the predicate and hence the ancestor's matches needn't be tested
- if a descendant tuple t_d does not satisfy the predicate, before dropping t_d its direct matches must be propagated to the tuple t_a of its immediate ancestor. The matches are needed, as they might satisfy predicate operators higher up in the plan.

These requirements suggest a natural evaluation strategy, which exploits and preserves the postorder sorting of the inputs, leading to full compositionality of all operators. The strategy calls for considering descendants first, using a stack to propagate matches to ancestors, as detailed in Algorithm 3. The stack tuples have schema (T, D) where T is an SCU tuple and D a boolean flag. The algorithm maintains the invariant that D is set to true iff the predicate is satisfied by $T.N$ or any of its descendants consumed so far from the input. When a new tuple s is read, if $s.N$ is an ancestor of the node at the stack top $top().T.N$, we cannot expect further input to contribute any descendants of $top().T.N$, and it is safe to pop (line 16). If the popped D flag is set to true, we record that s satisfies the predicate (line 11) through its indirect matches, so the predicate need not be checked on the direct matches of s (line 18). Otherwise, we drop the descendant (but not before propagating its matches to s (line 14)) and we check the predicate on s (line 19). If s satisfies the predicate either through direct or indirect matches, it is output. Either way, s is pushed on the stack (line 24) together with the verdict on the predicate's satisfaction, for subsequent consumption by ancestors of $s.N$.

EXAMPLE 3.4. *The query plan on Figure 6 illustrates the input and output* SCU *tables of Example 1.1. For instance, node* `legis-session` *(Dewey* 1.3*) is selected due to the propagation of its* `Jefferson` *match* 1.3.2.1.72 *by the* **ordered**$^{\exists}$ *predicate. The final set of answers is* `action`, `legis-session` *and* `bill`.

Algorithms Complexity. For each tuple consumed from the input, Algorithm 2 performs constant-time stack manipulation operations, computes an LCA (which depends on the length of the common prefix of the Dewey ids), and later, upon popping the tuple, it generates matches. When applied to inputs $|S_1|$ and $|S_2$ and producing output $|S|$, the running time contains: a linear component in the size of the inputs $|S_1| + |S_2|$; a linear component in the size of the output $|S|$; denoting with D the maximum nesting depth of a discovered LCA, it performs for each LCA at most D operations of scanning the Dewey id. The total number of LCAs is upper-bounded by the smallest size among $|S_1|$ and $|S_2|$: $O(|S_1| + |S_2| + |S| + D \times min(|S_1|, |S_2|)$. Algorithm 3 runs in time worst-case linear in the size of the input, $|S|$, if the predicate is not satisfied and all matches end up being inspected and propagated up the XML hierarchy.

Scoring. It is easy to see that the use of a stack in Algorithm 3 provides direct access to the descendants of a node, found at the top of the stack when the node gets pushed. This is compatible with the incremental computation of the score of a node from its descendant nodes. The node's score can be easily updated/recomputed as long as the scoring function depends on descendants only.

Algorithm 3 SCU Implementation of Full-Text Predicates

```
Require: $S$ is an SCU table sorted in postorder on $N$
    the predicate $P \in \{\textsf{ordered}^{\exists}, \textsf{window}^{\exists}, \textsf{dist}^{\exists}\}$ applies to individual SCU tuples
Ensure: outputs SCU table corresponding to
    $\texttt{reduce}(\sigma_P(\texttt{expand}(S)))$ sorted in postorder on $N$
initialize stack; open cursor on $S$
$s \leftarrow$ get_next($S$); $satisfies \leftarrow P(s)$
if ($satisfies = true$) then
    output $s$
end if
push($s, satisfies$)
while (not $EOF(S)$) do
    $s \leftarrow$ get_next($S$); $satisfies \leftarrow false$
    while ($s.N$ is ancestor of $top().T.N$) do
        if ($top().D = true$) then
            $satisfies = true$
        else
            {propagate matches to ancestor:}
            $s.M \leftarrow s.M \cup top().T.M$
        end if
        pop()
    end while
    {only check predicate if descendants violate:}
    if ($satisfies = false$) then
        $satisfies \leftarrow P(s)$
    end if
    if ($satisfies = true$) then
        output $s$
    end if
    push($T = s, D = satisfies$)
end while
```

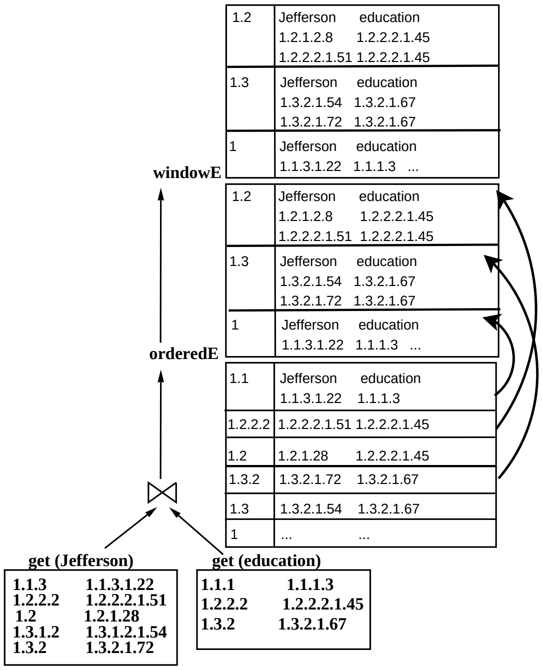


Figure 6: SCU Execution for Query in Example 1.1

4. EXPERIMENTS

Our experiments demonstrate the superiority of SCU over AllNodes . We also perform a detailed study of the SCU algorithm which shows that the overhead of using a stack and propagating matches is minimal. Finally, the comparison of SCU with GalaTex [12] a conformance implementation of XQuery Full-Text [28], and the TeXQuery Quark implementation [3] confirms our optimizations.

4.1 Experiments Setup

We implemented our algorithms in Java and ran experiments on a Centrino 1.8GHz laptop with 1GB of RAM. We used both real datasets, a 300MB DBLP document,[4] and synthetic documents generated with XMark.[5] Document sizes range from 50 to 300MB. Queries contain up to 5 joins and 4 full-text predicates. All times are reported in milliseconds and correspond to query execution times that do not include the time to access inverted lists. All queries used in the experiments are given in the table below.

Q	Description
q_1	**get**(`See`) **and get**(`internationally`)**and** **get**(`description`) **and get**(`charges`) **and** **get**(`ship`)
q_2	$\sigma_{orderedE}(\texttt{See,internationally,description,charges,ship})\, q_1$
q_3	push selections in q_2 at each join
q_4	$\sigma_{window_{>1}}(\texttt{See,internationally,description,charges,ship})\, q_1$
q_5	$\sigma_{window_{>90000000}}(\texttt{See,internationally,description,charges,ship})\, q_1$

When using document sizes with XMark, term frequencies varied as shown in the table below (we do not show the frequencies for all sizes). These frequencies are higher than previously reported frequencies [17, 24, 29] and affect directly the size of the inverted lists and of intermediate query results.

Term	50MB	100MB	200MB	300MB
`See`	3546	7242	14549	21670
`internationally`	3536	7081	14285	21260
`description`	3835	7847	15767	23503
`charges`	5662	11460	23097	34407
`ship`	5817	11709	23608	35166

4.2 Comparison of AllNodes and SCU

Varying Document Size. Figures 7 and 8 shows the result of running AllNodes and SCU algorithms on a conjunctive query q_1 and on two queries with predicates (q_2 and q_3) where the predicate is pushed in one case (q_3) and on XMark documents ranging from 50 to 300MB. Both algorithms grow linearly in document size with all 3 queries. On average, SCU has a 30% speedup over AllNodes . Note that on a 150MB document, AllNodes on q_3 beats SCU on q_2. However, SCU quickly catches up and the difference increases with larger documents due to the fact that AllNodes processes more entries, even if element nesting remains the same. It is not surprising to see that performing a relational-like rewriting improves performance by an average of 40%. A more interesting observation is that the improvement on SCU is more dramatic since the selection is pushed all the way down to each join and thus reduces the size of intermediate results early on.

4.3 SCU Performance

Varying Query Predicates. Figure 9 shows how SCU performs on queries q_1, q_4 and q_5 on an increasing document size. The predicate used in q_4 is always true and has not been pushed. It always finds that descendant nodes satisfying it, so its evaluation only introduces the overhead of

[3] *http://www.cs.cornell.edu/database/quark*

[4] *http://dblp.uni-trier.de/xml/*

[5] *http://monetdb.cwi.nl/xml/*

stack maintenance. This overhead, quantified as the difference between q_4 and q_1's times is very small and grows very slowly with larger documents since element nesting does not change much. The predicate in q_5 is always false and requires to propagate matches to ancestors. This is why the overhead, quantified as the difference between q_5 and q_1's times, is more important than the previous one. However, the good news is that it also grows very slowly with document size as element nesting does not grow.

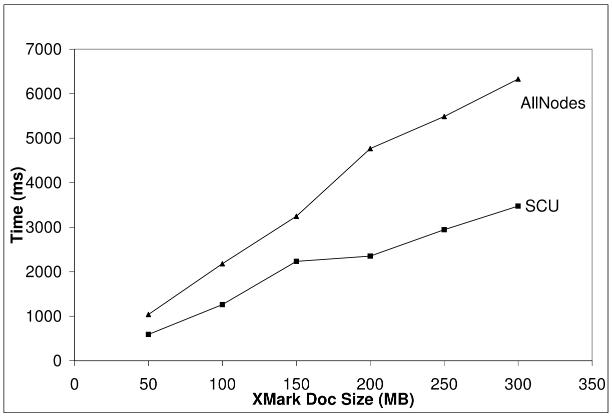


Figure 7: Vary Document Size (Query w/o Preds)

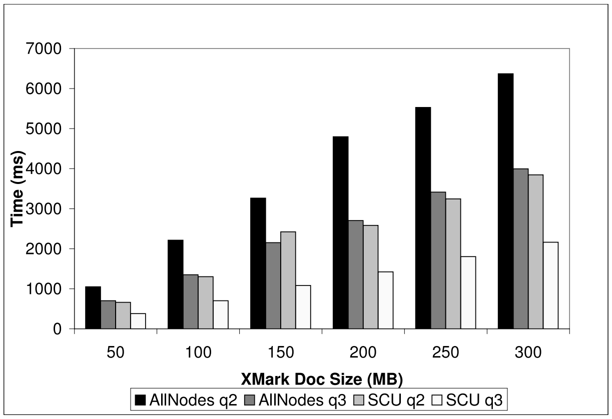


Figure 8: Vary Document Size (Query with Preds)

Varying Query Terms on Flat DBLP Data. Figure 10 reports running query q_1 with a varying number of terms (from 2 to 5 terms) on a 300MB DBLP document. Due to the fact that DBLP is very flat, AllNodes performs better than SCU since SCU pays the overhead of manipulating a stack. The good news are that the difference between the performance of the two algorithms does not get larger with queries containing more terms. Moreover, the time for AllNodes to read inverted lists and build input tuples is 3 seconds longer than the time for SCU, which manipulates shorter lists by definition.

4.4 Comparison with Existing Systems

We compared SCU with GalaTex and Quark (Figure 11). Due to the limitations of these two systems, we ran conjunctive queries (1 to 4 terms) on a 150KB XMark document. GalaTex and Quark have similar performances which are worse than AllNodes and SCU . The performance difference increases with queries containing more terms.

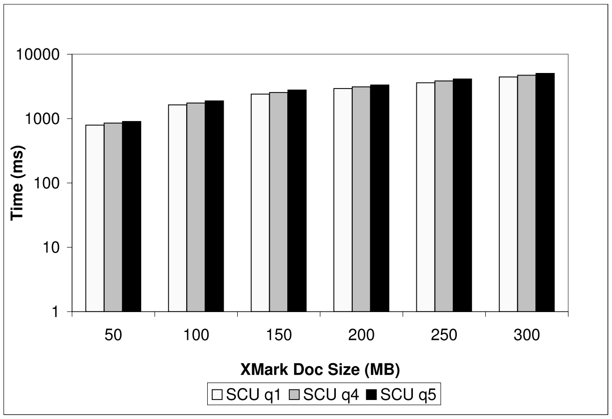


Figure 9: Vary Query Predicates

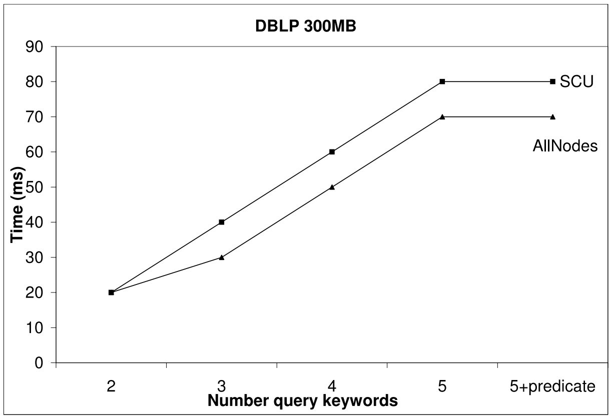


Figure 10: Vary Query Terms

5. RELATED WORK

Several full-text algebras and query evaluation algorithms have been proposed in the past [1, 11, 15, 18, 23, 27]. The most notorious algebras are the text region algebras which were proposed to model structured full-text search [9, 11,

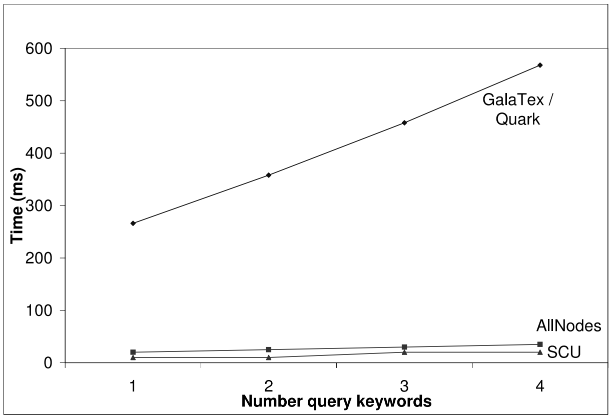


Figure 11: AllNodes , SCU , GalaTex and Quark

20, 23]. A text region is a sequence of consecutive words in a document and is often used to represent a structural part such as section and chapter. However, this algebra has limited expressive power [11]. The TIX and TOSS algebras [1, 18] focus on evaluating conjunctive queries and phrase search while the algebra in [27] is used to express NEXI queries [26].

There has been extensive research in information retrieval on the efficient evaluation of full-text queries [3], including structured full-text queries [6] and of XML queries such as XQuery/IR [5], XSEarch [10], XIRQL [14], XXL [25] and Niagara [30]. However, these works develop algorithms for specific full-text predicates in isolation.

The idea of computing the most specific elements for conjunctive queries has been actively explored using LCAs [17, 21, 24, 29]. We show in Section 3 that extending this idea to support the efficient evaluation of queries with complex full-text predicates needs to account for individual term matches in XML elements and, sometimes propagate matches along the XML hierarchy. Moreover, we show that a blind application of state of the art stack-based algorithms [17, 21, 29] results in higher complexities. This is due to the fact that preorder is a natural choice for XPath evaluation, since any other ordering would require materializing the document in main memory, or a two pass algorithm. However, for full-text, inverted lists are generated off-line and could be in postorder, which ends up being the natural order expected and automatically preserved by all our algorithms.

Relevance ranking methods for XML are based on extending the well-established vector and probabilistic methods [3] to incorporate structure by propagating answer scores along the XML tree [14], considering overlapping elements [8], applying length normalization to paths [7] or computing tag or path-based term weights [2, 10, 16]. None of them accounts for query predicates to score answers and is thus not applicable to distinguish between the binding and the existential semantics.

6. CONCLUSION

We presented the XFT algebra and efficient evaluation algorithms that account for element nesting in XML document structure to evaluate queries with complex full-text predicates. XFT subsumes the XQFT-class full-text languages, enabling a uniform treatment of their optimization and efficient evaluation problems. The novelty of our algorithms lies in their ability to combine relational query evaluation techniques with the stack-based exploitation of element nesting when evaluating full-text predicates. We are currently exploring the use of pull-based topk algorithms.

7. REFERENCES

[1] S. Al-Khalifa, C. Yu, H. V. Jagadish. Querying Structured Text in an XML Database. SIGMOD 2003.

[2] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, D. Toman. Content and Structure Scoring for XML. VLDB 2005.

[3] R. Baeza-Yates, B. Ribiero-Neto. Modern Information Retrieval. Addison-Wesley, 1999.

[4] A. Balmin, V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava, T. Wang. A System for Keyword Proximity Search on XML Databases. VLDB 2003.

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

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

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

[8] C. Clarke. Controlling Overlap in Content-Oriented XML Retrieval. SIGIR 2005.

[9] C. Clarke, G. Cormack, F. Burkowski. An Algebra for Structured Text Search and a Framework for its Implementation. Comput. J. 38(1): 43-56 (1995)

[10] S. Cohen, J. Mamou. Y. Kanza, Y. Sagiv. XSEarch: A Semantic Search Engine for XML. VLDB 2003.

[11] M. P. Consens, T. Milo. Algebras for Querying Text Regions: Expressive Power and Optimization. J. Comput. Syst. Sci. 57(3): 272-288 (1998)

[12] E. Curtmola, S. Amer-Yahia, P. Brown, M. Fernández. GalaTex: A Conformant Implementation of the XQuery Full-Text Language. XIME-P 2005.

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

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

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

[16] T. Grabs, H. Schek ETH Zürich at INEX: Flexible Information Retrieval from XML with PowerDB-XML. INEX Workshop 2002.

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

[18] E. Hung, Y. Deng, V. S. Subrahmanian. TOSS: An Extension of TAX with Ontologies and Similarity Queries. SIGMOD 2004.

[19] Initiative for the Evaluation of XML Retrieval. http://inex.is.informatik.uni-duisburg.de/2005/.

[20] J. Jaakkola, P. Kilpelainen. Nested Text-Region Algebra Report C-1999-2, Dept. of Computer Science, University of Helsinki, January 1999

[21] Y. Li, C. Yu, H. V. Jagadish. Schema-Free XQuery. VLDB 2004.

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

[23] A. Salminen, F. Tompa. PAT Expressions: an Algebra for Text Search. Acta Linguistica Hungar. 41 (1-4), 1992

[24] A. Schmidt, M. Kersten, M. Windhouwer. Querying XML Documents Made Easy: Nearest Concept Queries. ICDE 2001.

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

[26] A. Trotman and B. Sigurbjrnsson NEXI, Now and Next. INEX 2004.

[27] J.N. Vittaut, B. Piwowarski, P. Gallinari. An Algebra for Structured Queries in Bayesian Networks. INEX 2004.

[28] The World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text. Working draft. http://www.w3.org/TR/xquery-full-text/.

[29] Y. Xu, Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. SIGMOD 2005.

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