FleXPath: Flexible Structure and Full-Text Querying for XML

Sihem Amer-Yahia
AT&T Labs–Research
Florham Park, NJ, USA

Laks V.S. Lakshmanan
University of British Columbia
Vancouver, CA

Shashank Pandit
IIT Bombay
Mumbaî, India

ABSTRACT

Querying XML data is a well-explored topic with powerful database-style query languages such as XPath and XQuery set to become W3C standards. An equally compelling paradigm for querying XML documents is full-text search on textual content. In this paper, we study fundamental challenges that arise when we try to integrate these two querying paradigms.

While keyword search is based on approximate matching, XPath has exact match semantics. We address this mismatch by considering queries on structure as a "template", and looking for answers that best match this template and the full-text search. To achieve this, we provide an elegant definition of relaxation on structure and define primitive operators to span the space of relaxations. Query answering is now based on ranking potential answers on structural and full-text search conditions. We set out certain desirable principles for ranking schemes and propose natural ranking schemes that adhere to these principles. We develop efficient algorithms for answering top-K queries and discuss results from a comprehensive set of experiments that demonstrate the utility and scalability of the proposed framework and algorithms.

1. INTRODUCTION

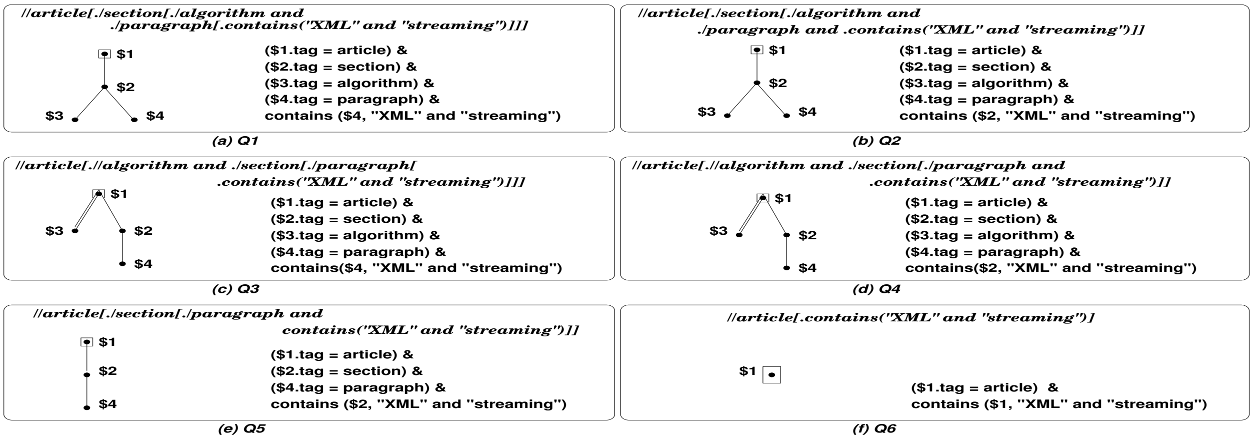
As businesses and enterprises generate and exchange XML data more often, there is an increasing need for searching and querying this data. Two major paradigms for searching XML documents are database style querying as exemplified by query languages such as XPath and XQuery, and IR-style querying, in particular, full-text and keyword search [28]. Ideally, users should not have to choose between these two paradigms but really benefit from both. Keyword search enhances the value of querying by permitting a fine level of querying textual content while query expressions, written in XPath or XQuery, bring value to keyword search by specifying a context in which to conduct the search. In addition, these languages allow the extraction of data at a very fine level of granularity, thereby returning to the user the most relevant document fragments in a document collection.

Several issues arise when attempting to put together these two querying styles. Should we use off-the-shelf XPath and IR engines or should we implement the integrated paradigm from scratch? We explore both cases. The first choice has the advantage of reusing existing techniques for XPath query evaluation and keyword search and looking for the best way to combine them for efficiency. The second choice has the benefit of modifying existing XPath evaluation strategies to better account for this integration. In both cases, we consider XPath expressions where a predicate might use the *fn:contains* function which looks for occurrences of specified keywords. The expression used in *fn:contains* can be as complex as an IR engine can handle (e.g., stemming, proximity distance, Boolean predicates). However, not all obligations specified in the XPath expression may be satisfied by a document, although it may be relevant for the *fn:contains* expression. A strict interpretation of the search context (i.e., the XPath query) would render many potential answers invalid and thus would penalize(!) the user for providing the context. Therefore, in order to leverage XPath in specifying the search context and, at the same time, not suffer from the consequences of the exact match semantics of XPath, we view the XPath expression on structure as a "template" and permit a flexible interpretation of this template. In other words, if an input document satisfies the XPath expression exactly, the requested answers will be returned. If an input document satisfies the expression only partially, it might be returned with a lower score. Query evaluation spans the space between the strictest (i.e., exact semantics) and the loosest interpretation of the XPath expression (i.e., the case where the only expression that is considered is the one specified in *fn:contains*).

We illustrate with an example the value and the inherent challenges in permitting flexible querying of XML documents combining both structure and keyword search.

Consider querying documents in the IEEE INEX data collection [1] or the ACM SIGMOD Record collection [2]. Such documents exhibit two desirable properties that we will explore in the rest of the discussion: heterogeneity in structure and presence of textual content. Figure 1 illustrates some example XPath queries. Every query in the figure is also shown in the form of a tree pattern together with a Boolean formula imposing constraints on nodes in the tree. Tree pattern queries constitute an important and expressive subset of XPath and make our illustration easier. Single edges denote parent-child containment, while double edges denote ancestor-descendant containment. Also, one of the nodes is a distinguished node (shown in a box in Figure 1), indicating that matches to this node are required as answers. All our queries in Figure 1 return articles.

Suppose a user wishes to find articles that are relevant to algorithms on streaming XML data. At one extreme, the user might simply issue the query $Q6$ of Figure 1(f), which asks for articles


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

(a) Q1

(b) Q2

(c) Q3

(d) Q4

(e) Q5

(f) Q6

Figure 1: Example Queries

containing the keywords "XML" and " streaming" *anywhere in the document*. This query is similar to IR queries and can be computed solely using IR techniques which include keyword and phrase search, proximity distance, stemming and thesauri. On the other hand, the user might wish to refine the scope of keyword search using some schema knowledge. Thus, the user might issue, say query $Q1$ of Figure 1(a). This query asks for articles containing a section subelement which contains an algorithm and a paragraph, such that the paragraph contains the keywords of interest. $Q1$ is more focused than $Q6$. E.g., $Q6$ may not distinguish between articles *containing* algorithms relevant to XML streaming and articles *mentioning* algorithms developed in other papers relevant to XML streaming without containing any algorithm themselves, but $Q1$ does.

However, suppose one of the documents queried contains an article which has a section containing an algorithm, the section title contains the keywords "XML" and " streaming", but none of the paragraphs in that section does. This article would be missed by query $Q1$ but might well be of interest to the user. A strict interpretation of $Q1$ means that the user who issued $Q1$ would not see such answers and would unfairly get "penalized" for providing useful context to direct the keyword search. Query $Q2$ (Figure 1(b)) would catch such articles because the *contains* predicate has been *moved up* from node $2 to its parent node $2. Thus, $Q2$ merely insists that section elements contain the keywords *anywhere*, but is otherwise identical to $Q1$. $Q2$ broadens the scope of application of the keyword search. In other words, $Q2$ creates a larger search context for the keyword search than in $Q1$. Therefore, all answers to $Q1$ also satisfy $Q2$. Similarly, an article that contains a section with a paragraph in that section containing the keywords, *with all algorithms being outside that section* will again be missed by $Q1$ (and by $Q2$), but might be of interest to the user. Query $Q3$ (Figure 1(c)) will catch such articles. The reason is that $Q3$ only insists that the article contains a (transitive) algorithm sub-element and a section that contains a paragraph containing the keywords of interest. The algorithm might be in that section, in a subsection of that section or, in another section. Note that $Q3$ includes all answers to $Q1$. Query $Q4$ combines $Q2$ and $Q3$. Query $Q5$ captures articles containing a paragraph sub-element which contains the keywords without having any condition on algorithm containment. The relationship between the five queries is: $Q1 \subset Q2$, $Q1 \subset Q3$, $Q2 \subset Q4$, $Q3 \subset Q4$, and $Q4 \subset Q5$. Finally, by repeatedly applying some primitive operations to the user query, once can produce query $Q6$ that contains all five queries. Intuitively, each query is a "relaxation" of the query it contains. The key point is that *if we adopt a strict interpretation of the user query $Q1$, many answers to one or more relaxations above, potentially of interest to the user's keyword search, would be missed out.*

One naive solution is for the user to write these queries by hand. This is both tedious and expensive, not only in terms of user time, but also in terms of the (potentially large) number of queries that the user might need to write and in terms of repeated processing of similar queries and, thus, of lost optimization opportunities. In this paper, we argue that the solution to this problem lies in treating the user query expression as a "template" and seeking answers that are approximate matches to this template, using a principled notion of approximation. We describe the FleXPath system that integrates XPath querying with full-text search in a flexible way, and make the following technical contributions.

- In order to integrate structure and keyword querying, we propose a formal framework in which queries on structure are viewed as a "template" for keyword search. Such queries are used to specify a context to conduct full-text search. In order to achieve this, we develop a query semantics that consistently extends classical semantics of queries without full-text search.

- A second question is how do we define approximate matches? Thereto, we formalize the notion of *query relaxation*. Intuitively, a relaxation to a query expression is any expression that contains the former. This admits a huge search space of relaxations, permitting many expressions that may be irrelevant to the user query. We provide an elegant definition of relaxation for the class of tree pattern queries (with full-text search) that addresses this problem.

- Thirdly, how do we span and search the space of relaxations? We present a set of primitive operations on queries that build on the ones proposed in [3, 15, 30], and show that they are independent (i.e., no operation can be derived from the others) and complete (they span exactly the space of relaxations defined). We define the semantics of a query so it includes answers to all relaxations of the query. We propose three natural schemes for ranking query results in this context and show that they satisfy certain desirable properties.

- A natural class of queries in this setting are top-K queries. We develop three algorithms for this purpose. Two of the algorithms (DPO and SSO) are designed to be able to use off-the-shelf XPath and IR engines while one (Hybrid) modifies an existing XPath evaluation algorithm. The algorithms can use any of the proposed ranking schemes and are designed to optimize repeated computation, the number of intermediate query answers, and the cost of (re)sorting answers to compute top-K results. We run experiments that evaluate the performance of the three algorithms.

In Section 2, we give some basic background and an overview of our problem. Section 3 defines relaxations and how to span the space of relaxations. In Section 4, we present our ranking schemes and their properties. Section 5 describes our query processing architecture and top-K algorithms. Section 6 contains the experiments carried on the algorithms. Related work is in Section 7.

2. BACKGROUND AND PROBLEMS

2.1 Tree Pattern Queries

We consider the class of *tree pattern queries*, an expressive fragment of XPath. A tree pattern query (TPQ) is a pair (T, F) where T is a rooted tree and F is a Boolean combination of value-based predicates. The nodes in T are labeled by variables, denoted $\$i$ where i is an integer. The edges are parent-child (pc) or ancestor-descendant (ad). Figure 1 contains examples of such queries. One of the nodes in T is designated as the *distinguished node* (shown inside a box in Figure 1) and identifies query answers.

Value-based predicates are of the form $\$i.tag$ =<tagname> that constrains the type of a node and $\$i.attr\ relOp\ value$. E.g., $\$i.price <$ 100 says the value of the *price* attribute associated with the element represented by node $\$i$ (say a book), must be < 100.[3] Note that pc-edges (resp., ad-edges) in the tree T are an inherent part of the query and formally correspond to assertions $pc(\$i, \$j)$ (resp., $ad(\$i, \$j)$). We call the latter predicates *structural predicates*. Thus, logically, the query should be understood as the conjunction of the formula F with all structural predicates $pc(\$i, \$j)$ and $ad(\$i, \$j)$ represented by T. For example, the logical expression corresponding to query $Q1$ in Figure 1 is given in Figure 2.

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

Figure 2: Logical Expression of Query $Q1$.

We define an additional value-based predicate $contains(\$i, \texttt{FTExp})$, that we allow in F. It takes a variable $\$i$ and a full-text expression `FTExp` and returns a Boolean value. The variable defines the *context* in which the full-text search expression given in `FTExp` operates. The predicate returns true if at least one node in $\$i$ satisfies `FTExp`. `FTExp` can vary from a simple conjunction of keywords to an expression that uses proximity distance, stemming, regular expressions and negation. In this paper, we do not focus on how to express such conditions in XPath. A language for such expressions is proposed in [2]. In the sequel, by tree pattern queries (TPQs), we mean TPQs with the *contains* predicate.

The semantics of a TPQ is captured in terms of a match. Let D be a data tree (i.e., an XML document collection) and $Q = (T, F)$ be a TPQ. A *match* is a function $f : Q \rightarrow D$ that maps the nodes of T to those of D such that: (i) all value-based predicates in F (including *contains*) are satisfied and (ii) all structural relationships are preserved, i.e., whenever $(\$i, \$j)$ is a pc-edge (resp., ad-edge) in Q, $f(\$j)$ is a child (resp., descendant) of $f(\$i)$ in D. Finally, the answer to a TPQ Q with distinguished node $\$d$, against an XML database D is the set of data nodes $Q(D) = \{x \mid x$ is a data node in $D \wedge \exists$ a match $f : Q \rightarrow D \wedge f(\$d) = x\}$.

Query containment is at the heart of relaxation. A TPQ Q is said to be contained in a TPQ Q', denoted $Q \subseteq Q'$, if for every XML database instance D, $Q(D)$, the result of applying Q to D, is contained in $Q'(D)$.

2.2 Problems Overview

Full-text search has been extensively researched in IR and has a semantics based on approximate match as a result of which query answers are ranked lists, unlike for conventional database queries. The first problem we tackle is how can we find a semantics for TPQs that is consistent with both database and IR paradigms. We argue that such a semantics should permit some degree of approximation of the XPath query. The second problem is how to score query answers under the new semantics. The third problem is how can we efficiently evaluate top-K queries ranked on the structural and full-text search expression(s). We address these three problems in the rest of the paper.

3. QUERY RELAXATION

3.1 Issues with Relaxation

Intuitively, a relaxation of a query is any query which contains the former. However, such a definition is too broad. Indeed, three principled ways of relaxing a query are (a) adding an explicit disjunction or union to the query, (b) replacing predicates present in the query by weaker ones, and (c) as a special case of (b), simply dropping those predicates. In this paper, we are not interested in relaxing the `FTExp` used in the *contains* predicate. So, we do not consider (b) further. A discussion on this case can be found in Section 3.4. Both (a) and (c) still have attendant drawbacks. E.g., consider query $Q1$ in Figure 1(a). Following (a), we can add an explicit union with an *arbitrary* query that asks for, say publisher addresses, thus permitting answers that are clearly irrelevant to the original query. Therefore, we regard relaxation by arbitrary union unacceptable. Following (c), we can drop one or more of the predicates $\$1.tag = \texttt{article}$, $contains(\$4,$ “XML” and “streaming”$)$ to obtain a relaxation. However, dropping the first admits non-articles as answers. Arguably, such answers may not be of interest to the user. Section 3.4 contains a discussion of this case and how such answers could be incorporated in a principled way. We do not consider them further here. Dropping the second predicate admits articles not containing the given keywords and thus, not relevant to the query. In addition, dropping an arbitrary predicate can lead to a query that is not a TPQ anymore. E.g., if we simply drop the predicates $pc(\$1, \$2)$ from the logical expression of query $Q1$ given in Figure 2, the result is a query whose pattern graph is disconnected. One can argue why not treat it as the union of two (or more) TPQs. The problem with this union is that the distinguished node in some of those trees is not well-defined. Furthermore, we wish to obtain relaxations of TPQs that are themselves TPQs.

In the next section, we show how we can avoid these pitfalls by reasoning on the logical expression of a TPQ.

3.2 Closure and Core of TPQs

Structural and value-based predicates in a TPQ imply other predicates. E.g., $pc(\$1, \$2)$ implies $ad(\$1, \$2)$. More generally, we have the inference rules shown in Figure 3.

The rules are self-explanatory. The last rule says if an element satisfies the full-text expression `FTExp`, then any element that (transitively) contains this element necessarily satisfies that expression.

[3] We do not consider “join” conditions that compare the contents/attributes of different nodes.

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

Figure 3: Inference Rules

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

Figure 4: Closure of Query $Q1$.

We define the *closure* of a TPQ as the expression obtained by taking the logical expression of the TPQ and adding (i.e., conjoining) all predicates that are derived using the above rules. The closure of query $Q1$ (Figure 4) is computed from its logical expression (Figure 2). It is easy to show that the closure of a TPQ is equivalent to it and is unique.

Given (a subset of) the closure C of a TPQ, a predicate p in C is said to be *redundant* if p can be derived from the rest of the predicates in C using the above inference rules. E.g., in the query $pc(\$1, \$2) \wedge ad(\$2, \$3) \wedge ad(\$1, \$3)$, the predicate $ad(\$1, \$3)$ is redundant. A query is minimal if it contains no redundant predicates. We define the *core* of a TPQ to be any minimal query that is equivalent to it. By the core of a TPQ, we mean the core of its closure. We can show:

THEOREM 1. **[Uniqueness of Core] :** Let Q be a tree pattern query. Then it has a unique core. ■

Flesca et al. [16] recently showed that when an XPath expression only uses child and descendant axes, wildcards, and branching, the minimal equivalent query can be obtained by pruning nodes and edges from it, thus showing that it has a unique minimal equivalent query. We can show that this result continues to hold when implication between predicates is taken into account. In other words, the closure of any TPQ has a unique core.

3.3 Defining Relaxations

We first deal with relaxations based on structural predicates. Let Q be a TPQ, C be its closure, and $S \subset C$ be a set of structural predicates. We denote the result of removing predicates S from C as $C - S$. *We assume that whenever a node variable $\$i$ does not appear in $C - S$, all value-based predicates involving $\$i$ are dropped automatically.*

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. ■

Intuitively, a structural relaxation of a TPQ is obtained by dropping predicates from its closure such that the resulting query has a core, itself a TPQ, that strictly contains the original query. Suppose we drop just the predicate $ad(\$1, \$3)$ from the closure $C1$ of query $Q1$ (see Figure 4), the resulting query is still equivalent to the original, since $ad(\$1, \$3)$ is derivable from the other predicates. So, $C1 - \{ad(\$1, \$3)\}$ is not a (structural) relaxation. Suppose we drop the predicates $pc(\$2, \$3)$ and $ad(\$2, \$3)$ from $C1$. The core of $C1 - \{pc(\$2, \$3), ad(\$2, \$3)\}$ is shown in Figure 5. This query corresponds to query $Q3$ in Figure 1. Clearly, $Q3$ is a TPQ and $Q1 \subset Q3$. So, $C1 - \{pc(\$2, \$3), ad(\$2, \$3)\}$ is a valid relaxation. Indeed, each of the queries in Figure 1(b)-(e) is a valid structural relaxation of query $Q1$ in Figure 1(a).

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

Figure 5: Core of Query $C1 - \{pc(\$2, \$3), ad(\$2, \$3)\}$.

It is important to note that structural relaxations *cannot* be defined on the basis of dropping predicates from a TPQ. *It is indeed necessary to consider the closure of the TPQ.* For example, in Figure 1, queries $Q3$ and $Q4$ cannot be obtained from $Q1$ in this way, even though they are relaxations of $Q1$.

In addition to structural relaxation, we consider relaxations on the *contains* predicate. While many forms of relaxations can be defined on value-based predicates,[4] we only consider this relaxation because it interacts with structure in in teresting ways.

DEFINITION 2. **[*contains*-Relaxation]** Let $Q = (T, F)$ be any tree pattern query and 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. ■

From the definition, it should be clear that a *contains*-relaxation of a TPQ is itself a TPQ and strictly contains the original query. As an example, in Figure 1, query $Q2$ is obtained from $Q1$ by relaxing the $contains(\$3,$ "XML" and "streaming"$)$ to $contains(\$2,$ "XML" and "streaming"$)$. Note that *contains*- and structural relaxations can be composed leading to various relaxations of the original query. For example, queries $Q5$ and $Q6$ in Figure 1, are obtained by applying a series of structural and *contains* relaxations to the original user query $Q1$. In the sequel, by the *space of relaxations* of a TPQ, we mean the space consisting of the TPQ as well as all its relaxations.

3.4 Other Relaxations

It is possible to consider other forms of relaxations [3, 15, 30]. E.g., if we have a type hierarchy associated with element types, then we can relax a query by replacing a tag with a tag associated with a supertype: e.g., in $Q1$, replace $\$1.tag = \texttt{article}$ with $\$1.tag = \texttt{publication}$ if the type hierarchy says `article` is a subtype of `publication`. We could replace value-based predicates, e.g., $\$i.\texttt{price} \leq 98$ with $\$i.\texttt{price} \leq 100$. We could also relax the *contains* predicate by making use of thesauri and replacing keywords with more general ones or drop some of the keywords. While these other forms of relaxations are interesting and useful, they are orthogonal to the ones studied in this paper. In fact, all relaxations that can be applied on the `FTExp` in the *contains* predicate can already be performed by a separate IR engine before returning its results. The goal of our relaxations is to *broaden the scope of full-text search* provided in the original user query and study its impact on query results.

3.5 Spanning Relaxations

Given a user query, we would like to consider the space of relaxed queries, evaluate them and return a ranked list of answers. A *significant challenge in this exercise is how to search this space.*

[4] E.g., a predicate $\$i.content > 5$ can be relaxed to $\$i.content > k$, for *each* number $k < 5$.

In particular, *contains*-relaxations on a TPQ with one or more *contains* predicates are easily enumerated by promoting *contains* to ancestors of the nodes they originally apply to. However, as examples from the preceding section show, identifying structural relaxation that can be applied to a user query, can be challenging. Dropping some predicates from an original TPQ('s closure) can violate the tree property or it can lead to a query that is equivalent to the original query. Ensuring that the resulting query is a structural relaxation involves a query containment check, a problem that is NP-hard in general [24]. What we would like is a systematic way to generate queries that are guaranteed to be relaxations and which covers all relaxations in the space. In this section, we present a set of operators for this purpose. The operators preserve the containment property between a query and its relaxed version. The first three operators are specific to querying structure and are similar to the ones proposed in [3, 15, 30]. The last one is new and is specific to querying text in XML documents.

3.5.1 Axis Generalization

The intuition behind this operator is that in place of a parent-child relationship between two nodes in a TPQ, if an ancestor-descendant relationship is present between these nodes in an actual database, they will be considered as a match. More precisely, for a TPQ $Q = (T, F)$ and a predicate $pc(\$x, \$y)$ in the logical representation of Q, $\gamma_{pc(\$x,\$y)}(Q)$, the *axis generalization* of Q on $pc(\$x, \$y)$, is the TPQ which is identical to Q except the pc-edge from $\$x$ to $\$y$ in T is replaced with an ad-edge from $\$x$ to $\$y$.

3.5.2 Leaf Deletion

The intuition behind this operator is that if we delete a leaf node $\$x$ in the query, we allow answers where that leaf node might not be matched. More precisely, given a TPQ $Q = (T, F)$ and a leaf node $\$x$ in T, $\lambda_{\$x}(Q)$, the leaf deletion of Q on $\$x$, is the TPQ Q' which is identical to Q except: (i) the leaf node $\$x$ is deleted from T, (ii) all value-based predicates involving $\$x$ are dropped from F, and (iii) if the deleted node $\$x$ is a distinguished node in Q, then the parent of $\$x$ in Q is made the distinguished node in Q'. As an example, $\lambda_{\$3}(Q2)$ applied to query $Q2$ in Figure 1 would result in a query $Q5$ where the leaf \$2 is deleted and the condition $\$2.tag = \texttt{algorithm}$ is dropped (i.e., replaced by "true") from $Q2$. In order to avoid queries that evaluate to true on every element, we forbid deleting the root of a TPQ. Query $Q6$ is an extreme case where leaf node deletion is applied repeatedly on the user query.

3.5.3 Subtree Promotion

The intuition behind this operator is that rather than insist on the existence of paths that go through nodes of a certain type, we simply insist on the existence of the said paths. More precisely, let Q be a TPQ, $\$x$ any node of Q other than the root, and $\$y$ its grandparent in Q. Then $\sigma_{\$x}(Q)$, the subtree promotion of Q on $\$x$, is the TPQ Q' identical to Q except the subtree rooted at $\$x$ is made a corresponding subtree of $\$y$, where the edge between $\$y$ and $\$x$ is an ad-edge. As an example, $\sigma_{\$3}(Q1)$ applied to query $Q1$ results in the query $Q3$ in Figure 1(c).

3.5.4 "contains" Promotion

The *contains* predicate, when imposed on a node $\$x$ of a TPQ, requires any match to contain the specified keywords somewhere within its scope. In *contains* promotion, we move this predicate from node $\$x$ to its query parent. More precisely, for a TPQ Q and a node $\$x$ in Q other than the root, $\kappa_{\$x}(Q)$, the *contains* promotion of Q on node $\$x$, is the TPQ where any *contains* predicate of the form $contains(\$x, \texttt{FTExp})$ is replaced by $contains(\$y, \texttt{FTExp})$, where $\$y$ is the parent of $\$x$ in Q. As an example, $\kappa_{\$4}(Q1)$ results in the query $Q2$ in Figure 1(b). This query admits sections containing the specified keywords somewhere, not necessarily in a paragraph in the section.

THEOREM 2. **[Soundness and Completeness] :** Let Q be a tree pattern query. Every query 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. ■

While we omit the proof for lack of space, soundness is intuitively clear. The intuition behind completeness is that *each valid relaxation can be regarded as obtained by dropping some predicates from a TPQ that cannot be derived from the remaining predicates.* We can identify the operator or sequence of operators that exactly captures each such dropping operation. The main value of this theorem is that it tells us that in the set of operators presented, we have a mechanism to systematically generate all and only valid relaxations of a given tree pattern query, while avoiding expensive minimization of a closure in order to obtain the core and expensive containment tests. When describing our algorithms, we exploit this correspondence between relaxation operators and predicate dropping: we often refer to "the next predicate dropped" when presenting our algorithms, for convenience and clarity, even though the algorithms are based on the operators.

4. DESIRABLE RANKING SCHEMES

4.1 Ranking on Structure and Keywords

While we believe a single ranking scheme may not always be acceptable, we identify some general principles that any good ranking scheme on both structure and keyword should adhere to.

Numerous algorithms have been proposed in the IR community (e.g., see [28]) for ranking documents based on keyword search. Our intention is not to propose yet another ranking algorithm for keyword search but rather focus on how to combine structural and keyword scores to produce the score of an answer. One can associate two numerical scores with an answer to a (relaxed) query: (i) a score reflecting how well it structurally matches the original query and (ii) a score based on its full-text expression. We regard these two scores as orthogonal and propose three alternative ranking schemes to combine these scores. This is different from existing content and structure ranking schemes in IR that rely on pre-specified XML fragments [9, 18]. We discuss these works in Section 7. However, a study and comparison of these ranking schemes is outside the scope of this paper.

Since we do not consider relaxations based on value-based predicates, we will assume they are satisfied when computing scores. A ranking scheme may associate a weight with each predicate present in the query. This weight may be user-specified, or computed by analyzing the input document. It may be static or dynamic (e.g., different for different elements with the same tag, depending on their context). Also, this weight may or may not depend on the query in question. For the *contains* predicate, we assume a weight of 1. We also assume the score returned by the IR engine for *contains* is normalized to be in the range $[0, 1]$.

DEFINITION 3. **[Answer Score]** Given a TPQ Q, the score of an answer to Q and, to any of its relaxations, is obtained by a computable arithmetic function of the weights associated with those query predicates (in the closure of Q) which are satisfied by the answer. ■

The score of an answer measures the relevance of that answer to the user query.

DEFINITION 4. **[Top-K Answers]** *Given a TPQ Q and its closure C, the top-K answers to Q and, to any of its relaxations, is the set of answers to Q and, to any of its relaxations, with the K highest scores.* ■

4.2 Properties of Ranking Schemes

Recall that every answer to a query is also an answer to any of its relaxations. Suppose Q is a (possibly relaxed) query and Q' is any relaxation of Q. We identify the following desirable properties for ranking schemes.

1. **Relevance Scoring**: Whenever a (possibly relaxed) query Q is relaxed to obtain Q', every answer of Q should have a structural score higher or equal to that of any answer of Q'. This is natural since answers that did not make the "first cut" imposed by Q are not as relevant as answers that exactly match Q. Note that keyword score may not satisfy this property: e.g., promotion of the *contains* predicate from a node to its query parent widens its search scope and may well result in a higher keyword score.

2. **Order Invariance**: A relaxation may be obtained by dropping predicates from a query('s closure) in any order. The score assigned to an answer to a relaxed query should be independent of the order in which predicates were dropped from an initial query to obtain the relaxed query.
   This property is orthogonal to the *means* used for generating relaxations. For example, in place of our operators from Section 3.5, one could use any other complete set of operators as long as this property is satisfied.

3. **Efficiency**: Finally, it is desirable that the ranking scheme used allow efficient query answering.

The first two properties are motivated primarily by semantical considerations while the last one is motivated by practical considerations. The first property is satisfied by making sure that the structural score of an answer decreases with the number of predicates that are dropped from the original query to evaluate that answer. The following theorem gives a sufficient condition for ensuring the second property.

Let f be any aggregate function, i.e., any total function from finite sets of multisets over real numbers to real numbers.

THEOREM 3. **[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. ■

Intuitively, the theorem says that if scores of answers are computed by a ranking scheme that combines weights associated with predicates satisfied by an answer using any aggregate function, then the ranking scheme is guaranteed to be order invariant. In particular, note that the aggregate function used may be arbitrary – i.e., distributive (like sum), algebraic (like average), or holistic (like median). The intuition is that the weight associated with each predicate does not depend on which predicate is present in a relaxation. Thus, the score assigned to an answer to a relaxation does not depend on how that relaxation was obtained, only on the predicates present in it. In practice, the structural score ss and keyword score ks may be computed separately by separate engines.

4.3 Ranking Schemes

In this section, we propose some example ranking schemes that adhere to the principles defined in Section 4.2. First, though we discuss "predicate penalty", a notion that is at the heart of score computation for relaxed queries.

4.3.1 Predicate Penalty

Given a TPQ Q and its closure C, the *predicate penalty* associated with each predicate p in C measures how much context an answer loses by not satisfying that predicate. A subtlety is that we must associate weights not just with predicates present in a query but with all predicates in its closure.

Suppose that each predicate in the closure of Q (including the *contains* predicate and excluding other value-based predicates) is assigned a weight using some function w_Q. All relaxations of Q correspond to dropping either the pc or the ad predicates or the *contains* predicate from a query closure. Consider a pair of nodes $\$i, \$j$ in Q. Suppose $\$i$ and $\$j$ are constrained to have tags t_i and t_j respectively. For tags t_i, t_j, we denote by $\#_{pc}(t_i, t_j)$ (resp., $\#_{ad}(t_i, t_j)$) the number of pairs of nodes of type (t_i, t_j) related by parent-child (resp., ancestor-descendant) relationships. We denote by $\#(t)$ the number of elements of tag t in the document and by $\#_{contains}(\$i, \texttt{FTExp})$, the number of matches to the `FTExp` expression in the context of node $\$i$.

We define the penalty associated with dropping the predicate $pc(\$i, \$j)$ from Q (but not $ad(\$i, \$j)$) as:

$$[\#_{pc}(t_i, t_j)/\#_{ad}(t_i, t_j)] \times w_Q(pc(\$i, \$j))$$

The intuition is that this penalty measures the loss of context resulting from relaxing pc to ad. If a majority of ancestor-descendant (t_i, t_j) pairs in a document are parent-child pairs, then the relaxation enables fewer additional answers than if this had not been the case. Thus, such a relaxation incurs a heavier penalty (i.e., closer to the weight of the edge itself).

We define the penalty of dropping $ad(\$i, \$j)$ from Q to be:

$$[\#_{ad}(t_i, t_j)/(\#(t_i) \times \#(t_j))] \times w_Q(ad(\$i, \$j))$$

This penalty is proportional to the ratio of the number of ancestor-descendant (t_i, t_j) pairs over the total number of elements of tag t_i or t_j. The more this ratio, the closer the penalty to the predicate weight.

Suppose $\$l$ is the parent node of $\$i$ in Q. We define the penalty of dropping $contains(\$i, \texttt{FTExp})$ (which corresponds to promoting it to $\$l$) as:

$$[\#_{contains}(\$i, \texttt{FTExp})/\#_{contains}(\$l, \texttt{FTExp})] \times w_Q(contains(\$i, \texttt{FTExp}))$$

The parent node $\$l$ contains at least as many matches to the full-text expression `FTExp` as its child node $\$i$. The intuition behind this penalty is that it measures context broadening of keyword search.

4.3.2 Answer Score

Suppose Q' is a relaxation of Q. Let $\{p_1, ..., p_k\}$ be the structural predicates present in Q. Let S be the set of predicates that were dropped from the closure of Q to obtain Q'.[5] Then the struc-

[5] In an actual algorithm, we will use our relaxation operations instead of predicate droppings. However, the effect can be understood in terms of predicate dropping.

tural score of any answer to Q' is denoted ss and is calculated as: $\sum_{i=1}^{k} w_Q(p_i) - \sum_{p \in S} \pi(p)$, where $\pi(p)$ denotes the penalty assigned to dropping predicate p.

The keyword score of a query answer is denoted ks, and is defined as the the weighted sum of the scores associated by an IR engine with all the *contains* predicates satisfied by that answer.

The final score of an answer combines its structural score ss and its keyword score ks. We propose three general ranking schemes for combining these scores that conform to the principles defined in Section 4.2.

Structure first: The score of an answer is a pair (ss, ks).
Keyword first: The score of an answer is a pair (ks, ss).
Combined: The score of an answer is computed with an arithmetic function that combines ks and ss, e.g., the sum $ks + ss$.

When the combined score is an ordered pair, we use the standard lexicographic ordering. The above classification makes no commitment to *how* structural and keyword scores are computed. A specific ranking scheme is obtained by choosing a strategy for computing predicate penalties associated with relaxations, as in Section 4.3.1 and a general ranking scheme. The following example illustrates this.

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

$pc(\$1,\$2) \wedge pc(\$2,\$4) \wedge \$1.tag = \texttt{article} \wedge \$2.tag = \texttt{section} \wedge \$4.tag = \texttt{paragraph} \wedge ad(\$1,\$2) \wedge ad(\$2,\$4) \wedge ad(\$1,\$4) \wedge contains(\$2, \text{“XML” and “streaming”}) \wedge contains(\$1, \text{“XML” and “streaming”})$.

Figure 6: Closure of Queries $Q1$ and $Q5$.

EXAMPLE 1. **:** Consider the closure of $Q1$ and $Q5$ in Figure 6. $Q5$ is obtained from $Q1$ by promoting the *contains* predicate from node \$4 to its parent node \$2 (i.e., dropping the predicate $contains(\$4, \text{“XML” and “streaming”})$ from $Q1$('s closure) and then, deleting node \$3 (i.e., dropping predicates $pc(\$2,\$3)$, $ad(\$2,\$3)$, $ad(\$1,\$3)$). Suppose our weight assignment is uniform and assigns a unit weight to every structural and *contains* predicate in $Q1$. The structural score of an answer to $Q1$ is then equal to 3. The penalty of dropping the predicates to obtain $Q5$ is (Exp denotes the full-text expression in $Q1$):

$$(\#_{pc}(\texttt{section},\texttt{algorithm})/\#_{ad}(\texttt{section},\texttt{algorithm})) \times 1+$$
$$(\#_{ad}(\texttt{section},\texttt{algorithm})/(\#(\texttt{section})\times\#(\texttt{algorithm})))\times 1+$$
$$\#_{ad}(\texttt{article},\texttt{algorithm})/(\#(\texttt{article}) \times \#(\texttt{algorithm})) \times 1+$$
$$(\#_{contains}(\texttt{algorithm},\texttt{Exp})/\#_{contains}(\texttt{article},\texttt{Exp})) \times 1$$

The structural score of answers to $Q5$ is then 3 minus the penalty computed above. The final score of answers to $Q5$ is computed in the same manner as $Q1$, by combining their keyword score and structural score using one of the general ranking schemes above. ■

The ranking schemes proposed in this section all conform to properties 1 and 2 defined in Section 4.2. In the next section, we shall evaluate their efficiency (property 3) empirically by exploring how to put together an IR engine and an XPath engine and stydying how to optimize repeated computation, the number of intermediate answers and the cost of (re)sorting to compute the top-K answers.

5. QUERY PROCESSING

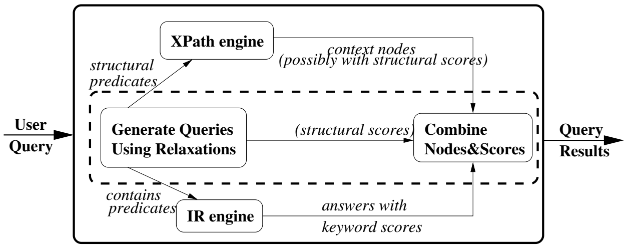


Figure 7: FleXPath Query Processing Architecture

5.1 FleXPath Architecture

Figure 7 depicts the general architecture of our top-K query evaluation in FleXPath. We developed three algorithms. All our algorithms assume that the *contains* predicate is evaluated by a separate IR engine that returns a ranked list of pairs (node,score). In our implementation (Section 5.2), we use the same techniques as in [20, 29] that return the most specific elements that satisfy the full-text expression used in *contains*. The output of the IR engine is combined with the output of the XPath engine by comparing the returned nodes from the IR engine with the context elements returned by the XPath engine.

We developed three algorithms – DPO, for Dynamic Penalty Order and, SSO, for Static Selectivity Order and Hyrid, a performance improvement over SSO. While DPO relies on evaluating multiple queries one by one to decide if an additional relaxation is needed, SSO uses selectivity estimates to decide which relaxations to encode in a query before sending that query (at once) to the XPath and IR engines.

Our algorithms assume that structural conditions are evaluated independently of any *contains* predicates. An alternative possibility would first use an inverted index to evaluate the *contains* predicates and filter out potential answers, and then match structural predicates. The efficiency of each approach depends on the types of queries. A comparison of these two approaches would be interesting but is outside the scope of this paper.

We provide the pseudo-code of our algorithms for the structure first *ranking scheme (see Section 4.3).* The pseudo-code would need to be modified to accommodate the other ranking schemes. For the keyword-first scheme, all relaxations need to be encoded in the query because an answer with the worst structural score might still make it to the top-K answer set.

When the Combined ranking scheme is used, we use the following pruning technique. Suppose $Q1, Q2, ..., Qn$ are the relaxations in descreasing order of structural score. Let the structural and keyword score of Qi be ssi, ksi respectively. Suppose i is the smallest number for which the set of answers to $Q1, ..., Qi$ equals or exceeds (exactly for DPO and according to estimates for SSO) K. Stopping after Qi might miss some of the top-K answers. Suppose there are m *contains* predicates in the query. Recall that *contains* has weight 1 and so the keyword score for each predicate is ≤ 1. Let $j \geq i$ be the smallest number for which the structural score ssj of Qj satisfies $ssj \leq ssi - m$. Clearly, no answer to relaxations $Qs,\ s > j$ can be in the top-K answer set so we can ignore all relaxations $Qs,\ s > j$.

5.1.1 DPO

The main benefit of DPO is the fact that it is able to use off-the-shelf XPath and IR engines. DPO relies on query-rewriting as in [15, 30]. However, instead of generating a potentially large number of queries, one for each relaxation, DPO decides which relaxations to consider, using predicate penalties. For brevity, we omit the pseudo-code and explain the algorithm for the structure-first case.

DPO admits a user query as input, computes its closure and sorts its predicates by increasing penalty order. Then, it evaluates the user query by sending it to the XPath engine. If the number of answers to that query does not exceed K, it drops[6] the predicate with the lowest penalty from the query, sends the resulting query to the XPath engine which computes and returns the results. Our predicate dropping ensures that answers to a query include answers to the previous query and no answer will be lost. For example, if the query is A [.//B and .//C] and if the predicate .//C is dropped, then at the next step, predicate .//B could be dropped but this does not mean that the query A[.//C] will never be considered since dropping corresponds to making predicates optional (and not loosing them entirely). This is captured by using vectors of answer lists for avoiding recomputation (see Section 5.2.2).

Based on the number of answers obtained at each step, DPO decides or not to further relax the query by dropping the predicate with the next lowest penalty. Thus, DPO might generate and send multiple queries to the XPath and IR engines. DPO has the property that answers to a query will have a lower score than answers to a preceding query. Therefore, it appends query results without having to sort results. The algorithm then calls an IR engine to evaluate the *contains* predicates in the user query and combines the result. When the number of answers exceeds K, DPO stops query evaluation.

If the number of relaxations to encode in the query is small, we expect DPO to perform well. Due to the fact that the non-relaxed version of a query is contained in its relaxed version, DPO might do a lot of repeated computation. We will see how we avoid this in our implementation in Section 5.2.

5.1.2 SSO

The pseudocode for SSO is given below. This algorithm decides which relaxations to keep in the query to obtain K answers, based on the selectivity estimates and the penalty of the predicates that were dropped to obtain the relaxation. For example, given the closure of query $Q1$ (see Figure 4), SSO might decide to drop the two structural predicates $ad(\$2, \$4)$ and $ad(\$1, \$3)$ because they have the lowest penalties (i.e., these relaxations produce answers with the highest scores) and the sum of the selectivities of the remaining predicates guarantees that at least K answers will be produced. In general, SSO needs a selectivity estimator that can provide a lower bound on the number of results for an XPath expression. If not, then potentially we might need to restart SSO to compute the top-K results.

SSO proceeds as follows. First, it computes the query closure (line 1) and sorts its predicates in increasing penalty order (line 2). Then, unlike DPO which evaluates the query, SSO estimates the result size of the query using techniques such as the ones in [27] (line 3). If the estimated number of answers is less than K, SSO drops the next predicate with the lowest penalty from the query (line 5). The process continues until the number of estimated answers is at least K. At this point, SSO decides to evaluate the structural part of the query (line 8). The IR engine is invoked to compute the *contains* predicates (line 9). Finally, SSO combines the results (line 10). When the selectivity estimation is not accurate, the number of answers might be smaller than K (line 11), in which case, the algorithm has to drop more predicates. When the number of answers exceeds K, the final result is sorted and pruned (line 15) and returned to the user.

Algorithm 1 Static Selectivity Order (SSO)

```
Require: Query Q, K
 1: closureQ = computeClosure(Q);
 2: currentQ = sortPenalties(closureQ);
 3: estimNumAnswers = estimResultSize(currentQ);
 4: while estimNumAnswers < K do
 5:     currentQ = dropNextPredicate(currentQ);
 6:     estimNumAnswers += estimResultSize(currentQ);
 7: end while
 8: resultQ = evaluateQuery(currentQ);
 9: IRresult = IRevaluate(FTExp(currentQ));
10: tempResult = Combine(resultQ,IRresult);
11: if ComputeSize(tempResult) < K then
12:     Goto 5;
13: end if
14: finalResult = SortandPrune(tempResult,K);
15: return finalResult;
```

5.2 Implementation

5.2.1 Join Plans for DPO and SSO

Both DPO and SSO could use an off-the-shelf engine for evaluating structural queries (the `evaluateQuery()` function in both pseudocodes). However, we chose to implement our own evaluation engine in the two cases for two reasons. First, DPO would benefit from a modification which avoids recomputing answers that are common to two successive queries. Second, SSO would benefit from a more accurate computation of answer scores. Unlike DPO, where all answers to a specific relaxation have the same structural score which is known at compile time (i.e., they all satisfy the same set of query predicates), answers to the query produced by SSO (which encodes several relaxations into one query) might satisfy a different subset of predicates which are known only at query evaluation time. This will become clear after we briefly present our evaluation algorithm.

We represent a query using left-deep join plans and we use the structural join algorithm given in [1]. This algorithm requires input lists to be sorted on node identifiers. Relaxations are encoded in the same join plan in the same manner as in [3]. Figure 8(a) shows the join plan corresponding to query $Q1$ given in Figure 1, without the *contains* predicate. Figure 8(b) contains the join plan of $Q3$, a relaxed version of $Q1$, and Figure 8(c) contains the join plan of $Q5$, another relaxed version of $Q1$. These plans group a structural predicate and its derived version (using the inference rules given in Figure 3) in the same join predicate. The structural scores of nodes returned by the join plan are computed as described in Section 4.3. Our join plans are not affected by the ranking scheme that we use. We omit the *contains* predicate for brevity.

We denote r_1, the relaxation that drops predicate $pc(\$2, \$3)$ from the closure of $Q1$ (given in Figure 4), r_2, the relaxation that drops $ad(\$2, \$3)$ from the closure of $Q1$, r_3, the relaxation that drops $contains(\$4, \mathtt{FTExp})$ from the closure of $Q1$. Suppose r_1, r_2, r_3 are ordered by increasing penalty, and that using selectivity estimates, SSO decides that all three relaxations need to be encoded in $Q1$ to obtain at least K answers. Then SSO would generate and evaluate the join plan of $Q5$ while DPO would first evaluate the

[6] Recall that predicate dropping is achieved using relaxation operations of Section 3.5.

join plan of $Q1$, find out if the number of answers is less than K, evaluate it, and so on.

5.2.2 *Implementing DPO and SSO*

In order to preserve DPO's benefit, we designed its algorithm in such as way that it saves computation. For example, suppose DPO evaluates first the query plan in Figure 8(a). It builds a vector for each predicate in the query. If the next join plan to evaluate is 8(b) (say, because there were not enough answers produced by the first join plan), then since the difference between the two plans is that node `algorithm` has been promoted, DPO excludes all `algorithm-section` pairs (a, s) where a is a child of s.

SSO relies on keeping a `threshold` at each join node to prune answers that will not end up in the top-K set, as early as possible in query evaluation. We refer to this threshold as the maximal score growth of an answer (`maxScoreGrowth`). It is associated with each join in the join plan and records the sum of the weights of the remaining predicates in the plan. Intuitiveley, it corresponds to the highest score an intermediate answer can have in the remaining plan. When an intermediate answer arrives at a join, SSO decides whether or not this answer should remain in the set of answers, based on its current computed score and the `maxScoreGrowth` at the join node. If the sum `threshold` + `maxScoreGrowth` is lower than the score of the current kth result, the intermediate answer is discarded. In order to know what the score of the kth result is, the set of intermediate query answers is sorted on their score. This algorithm is similar to the one proposed in [3]. The difference is that SSO uses selectivity estimates and penalties to decide which relaxations will be encoded in the query while in [3], all possible relaxations are initially encoded in the query thereby resulting in large intermediate query results.

The bottleneck of SSO is that the algorithm used to evaluate the structural join expects its result to be sorted on node identifiers while pruning intermediate query answers requires their sorting on scores. There is a fundamental tension between these two sort orders. We address this weakness in the next section.

5.2.3 *Hybrid*

We want to combine the best of DPO and SSO. DPO's strength is that it does not compute any relaxation unless at least one answer from that relaxation is guaranteed to end-up in the top-K set. SSO's strength is that it uses a single evaluation plan for computing all relaxations deemed necessary for the given top-K query, thus avoiding repeated passes over the data. However, SSO might resort data. We propose Hybrid, an algorithm that avoids resorting on scores using *bucketization*. The key idea behind Hybrid is to create buckets of intermediate results at each join where each bucket corresponds to a set of predicates. Answers in a bucket satisfy the same set of predicates and so have the same score. E.g., in the query plan for $Q1$ given in Figure 8, when computing $c(\texttt{article}, \texttt{algorithm})$ or if not $c(\texttt{article}, \texttt{algorithm})$ then $d(\texttt{article}, \texttt{algorithm})$, we would create two buckets for the output – one for results satisfying $c(a, b)$ and another for results satisfying
$d(\texttt{article}, \texttt{algorithm})$ but not $c(\texttt{article}, \texttt{algorithm})$. Within each bucket, answers are sorted on their node id. Since this sort order is preserved by the join algorithm we use, no additional sorting is necessary. We illustrate this in the pseudocode given below.

Algorithm 2 Hybrid

```
Require: Node n, k
 1: if (n is leaf) then
 2:   list = evaluateLeaf(n);
 3:   return list;
 4: end if
 5: list1 = Hybrid(n.left);
 6: list2 = Hybrid(n.right);
 7: for r1 in list1 do
 8:   for r2 in list2 do
 9:     s = computeScore(r1,r2,n.predicate);
10:     if ((s + n.maxGrowth) $\geq$ current_k) then
11:       res = computeResult(r1,r2,n.predicate);
12:       if ($\nexists$ bucket(r1,r2,n.predicate)) then
13:         b = createBucket(r1,r2,n.predicate);
14:       else
15:         b = getBucket(r1,r2,n.predicate);
16:       end if
17:       addToBucket(res,b);
18:     end if
19:     return current_buckets();
20:   end for
21: end for
```

Hybrid is a recursive algorithm that takes the root of a join plan and a value for K and, returns the top-K answers. If the input node is a leaf (line 1), it is evaluated (line 2) otherwise, its left (line 5) and right (line 6) subplans are sent to the algorithm. We show a nested loop join algorithm for simplicity of exposition. Given two input lists, the algorithm computes the score of an intermediate answer using the scores of its inputs and the score of the join predicate (n.predicate) that is used to compare the two inputs (line 9). If that score is not going to end up in the top-K answers (line 10), it is dropped, otherwise, Hybrid decides to create a new bucket for that answer (line 13) if a bucket with the same score does not exist (line 12) or, it puts it in an existing bucket with the same score (line 15). Depending on the predicate used to generate it, an intermediate result will end-up in a different bucket. The output of a join might be multiple buckets, one for each distinct score. Pruning of intermediate answers translates to elimination of buckets with a score (added to `maxScoreGrowth` at the join) lower than the score of the current bucket containing the kth answer.

DPO's strength comes from the fact that (i) it evaluates the latest relaxation determined as necessary, counts the answers, and then decides if the next relaxation is necessary, thus using exact knowledge; and (ii) it does not require resorts on intermediate results. However, the price is that it might make repeated passes over input lists. Hybrid minimizes the number of passes over input data (like SSO) while at the same time never sorting intermediate results on score (like DPO). Also, because of bucketization, we continue to have the advantage of SSO in pruning intermediate results using `threshold` and `maxScoreGrowth`. Note that buckets are ordered by score, since each bucket is uniquely identified by the set of structural predicates satisfied by the answers it contains, from which `threshold` can be calculated.

In the next section, we substantiate these claims empirically.

6. EXPERIMENTS

We ran several experiments to show the utility and scalability of our algorithms when relaxing the user query. Our experiments compare and evaluate the performance of DPO, SSO and Hybrid algorithms under various conditions.

Setup: We run the experiments on an Intel CPU P4 2GHz machine with 512MB memory and 10GB disk space under Linux Redhat version 7.2. Our programs were written in Java and C for the expat XML parser.

Dataset and Queries: We use the XMark XML data generator (`http://monetdb.cwi.nl/xml/index.html`). We varied the size of our documents from 1MB to 100MB. In order to "exploit the heterogeneity of the datasets" for relaxation, we designed three

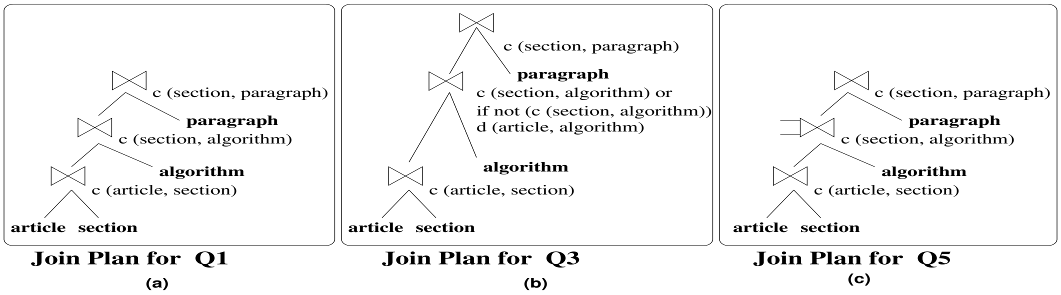


Figure 8: Join Plans

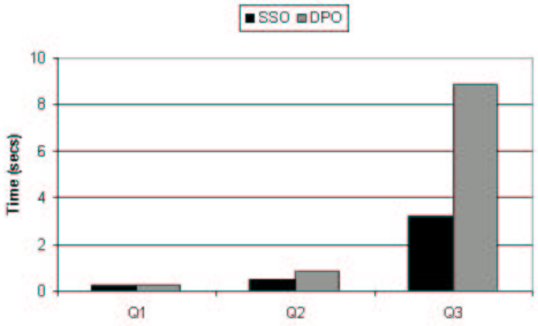


Figure 9: Varying Number of Relaxations

queries $Q1$, $Q2$ and $Q3$.

$Q1$: `//item[./description/parlist]`

$Q2$: `//item[./description/parlist and ./mailbox/mail/text]`

$Q3$: `//item[./description/parlist/listitem and ./mailbox/mail/text[./bold and ./keyword and ./emph] and ./name and ./incategory]`

Edge generalization is enabled by recursive nodes in the DTD (e.g., `parlist`). Deleting leaf nodes is enabled by optional nodes in the DTD (e.g., `incategory`). Finally, subtree promotion is enabled by shared nodes (e.g., `text`).

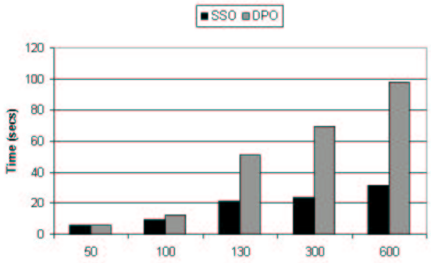


Figure 10: Varying K

Selectivity estimation is an integral part of the SSO algorithm. It allows SSO to statically estimate the number of relevant relaxations. Although we could use off-the-shelf selectivity estimators such as [27], we decided to build our own because the precision with which the estimation is done has a significant impact on SSO. In case, the estimation is not precise and the number of answers obtained are less than what is estimated, we would need to restart SSO. In our implementation, we have developed an estimation technique which gave precise estimations for our data sets. We first do intensive pre-processing of the document in order to obtain counts of the various types of nodes and edges in the XML document. We then assume a uniform distribution of elements. For example, suppose 60% of A's in the document have a B as a child. We assume that this fraction is independent of the location of A in the document. So we estimate that in 60% of occurences of C/A also, A will have B as a child. So, the estimate for C/A/B is 0.6 times that of C/A. We found that this technique works well for our dataset and we never had to restart SSO.

Parameters: In order to compare our algorithms we varied the size of the input documents, the size of K in top-K, the query size and the number of relaxations that a query can admit. $Q1$ admits one relaxation: generalize edge `description/parlist`. $Q2$ contains $Q1$ and admits promote `text`. $Q3$ contains $Q2$ and admits delete `incategory` and generalize edges `parlist/listitem`, `text/bold`, `text/keyword` and `text/emph`.

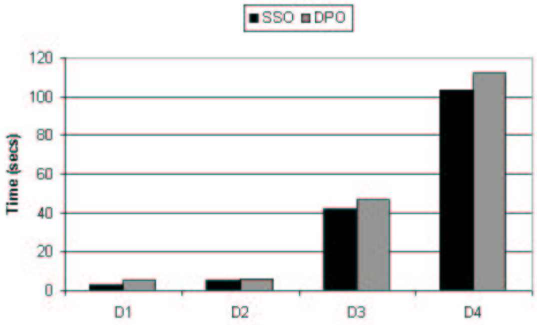


Figure 11: Varying Document Size (K = 12)

Preliminary results: We report four preliminary experiments that compare DPO and SSO. Figure 9 shows a comparison between DPO and SSO on a 1MB document, with K set to 50 answers. We report execution times for the three queries. The results show that by increasing the number of relaxations (no relaxation when running $Q1$, 2 relaxations when running $Q2$ and 6 relaxations when running $Q3$), SSO performs better than DPO and the difference between the two algorithms increases with the number of relaxations. Figure 10 reports the time of evaluating query $Q3$ using both algorithms on a 10MB document. K varies from 50 to 600 answers.

When the number of answers is small (50) SSO and DPO last the same time since no relaxation is required. When the number of answers increases, more relaxations are required and the number of intermediate answers increases. The effect of pruning in SSO

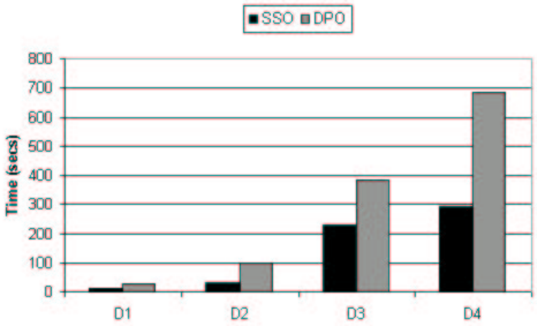


Figure 12: Varying Document Size (K = 500)

makes this algorithm superior to DPO, reaching 68% improvement in query time when the number of answers is 600. The last two experiments report a comparison between DPO and SSO when the size of the input document increases (from 1MB to 100MB). In the first one (Figure 11), K was set to 12 answers while in the second one K was equal to 500 answers (Figure 12). Both experiments are run on $Q2$. In the case where K is small, DPO and SSO are close because the only case where a relaxation is needed is on the 1MB document (on which `text` is promoted to `mailbox` in order to obtain at least 12 answers). In the case where K is large (500 answers), more relaxations are encoded in $Q3$ (from 0 to 8 relaxations). By increasing the number of relaxations and the size of the document, we increase the number of intermediate results and thus, the difference between DPO and SSO since SSO is better at pruning intermediate query answers.

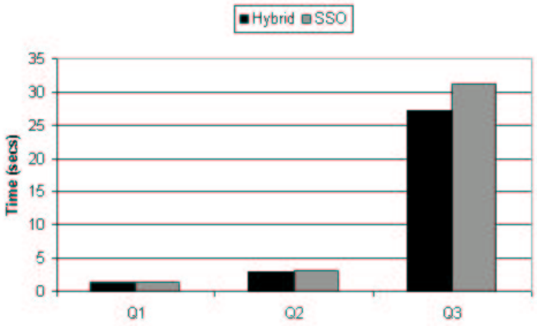


Figure 13: Varying Number of Relaxations (K = 500)

These first experiments show the scalability of SSO with respect to the number of relaxations, the number of query answers and the size of input documents. The difference between SSO and DPO plots increases with K.

SSO and Hybrid: The previous experiments showed the superiority of SSO over DPO. We will now compare Hybrid to SSO.

Overall, Hybrid does not do much better than SSO (although it does consistently better). However, the difference between the two algorithms increases with an increasing K or document size or number of relaxations. Figure 13 verifies this claim in the case where we varied the number of relaxations with a value of K set to 500 on a 10MB document.

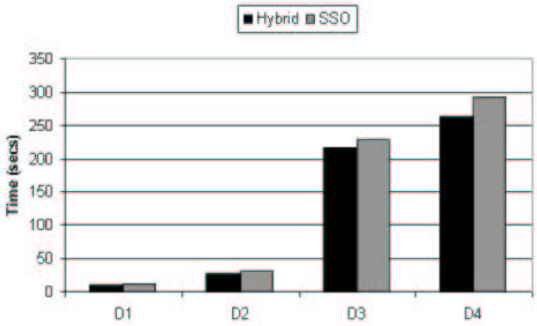


Figure 14: Varying Document Size (K = 500)

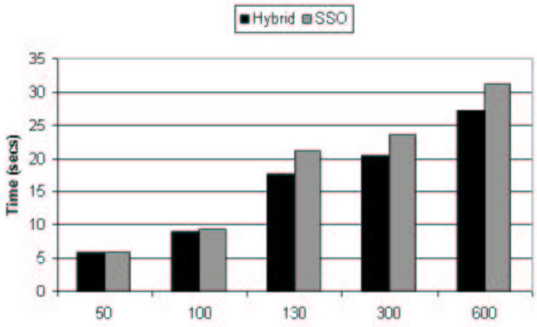


Figure 15: Varying K (DocSize = 10MB)

Figure 14 shows the case of running query $Q3$ with K set to 500 (1MB to 100MB documents). Figures 15 report the time to evaluate query $Q3$ on a 10MB and 100MB document respectively. The figures show that Hybrid is useful even in the case of small documents since SSO might be sorting large sets of intermediate query answers. The difference between the overall times of the two algorithms increases when increasing K even if document size is small. This is due to the fact that *SSO is more sensitive to the value of K than Hybrid* because the size of intermediate answers that need to be resorted depends on K.

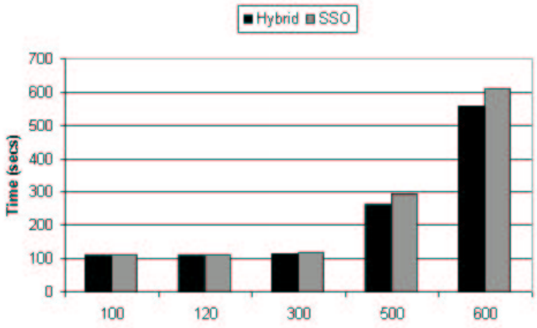


Figure 16: Varying K (DocSize = 100MB)

7. RELATED WORK

Combining structure and text search has attracted a lot of interest in the database and the information retrieval communities [4, 5, 6, 11, 12, 13, 17, 18, 19, 25, 26, 31]. In IR, approaches are classified to content-only search (CO) and content and structure search (CAS). CAS approaches include ELIXIR [11], XIRQL [18] and JuruXML [9]. These approaches focus on a vague matching of limited XPath predicates and on designing specific indices to score document fragments. For example, JuruXML [9] relies on element=-specific indexing ad on the vector-space model to score query results. However, it allows only limited XPath queries which must be known in advance to be pre-indexed.

Different relaxations on structure have been defined previously. [15] defines: unfold node, delete node, propagate condition at a node to its parent. In both [18] and [30], the authors define relaxations on XQL such as generalize datatypes, ontologies on elements, edit distance on paths, delete node, insert intermediate nodes and rename node. Finally, [3] proposed the relaxations: relax node, delete node, relax edge, promote node. Our work is the first formalization of relaxations on structure in XML queries and the first sound and complete algebraic framework for spanning relaxations. In addition, we propose ranking schemes and define properties that they must satisfy and develop efficient evaluation algorithms.

There are three evaluation strategies for approximate XML queries. *Rewriting strategies* such as DPO [11, 15, 18, 30] enumerate possible queries derived by transformation of the initial query. Some of these strategies use schema information to reduce the number of queries [30]. *Plan-based strategies* such as SSO and Hybrid [3, 23] encode relaxations in the query evaluation process. Finally, *Data-relaxation* computes a closure of the document graph by inserting shortcut edges between each pair of nodes in the same path [14]. Since this strategy was shown to quickly fail with large databases, we experimented the first two strategies in FleXPath.

In relational databases, Carey and Kossmann [8] optimize top-K queries when the scoring is done through a traditional SQL order-by clause, by limiting the cardinality of intermediate result tuples. Although algorithmically different, SSO is similar to works that use statistical information to map top-K queries into selection predicates (which may require restarting query evaluation when the number of answers is less than K [7, 10, 22]).

8. CONCLUSION

We presented FleXPath, a framework that integrates structure and full-text querying in XML. A key idea in FleXPath is to interpret the XPath query as a template for keyword search, thereby enabling approximate query answers on structure and using it to provide a context for full-text search. FleXPath is the first formalization of query relaxations that characterizes the space between the strictest (i.e., exact semantics) and the loosest interpretation of the XPath expression. We studied ranking schemes and developed algorithms for top-K queries that span this space and evaluated their performance. FleXPath initiates a wide range of new research opportunities including a tighter integration of structure and keyword indices for a more efficient approximate query evaluation.

Aknowledgments

The authors wish to gratefully acknowledge valuable comments from the anonymous reviewers. The second author's work was supported in part by grants from NSERC (Canada) and NCE/IRIS. Part of this work was performed when the third author was visiting UBC as an intern in Summer 2003.

9. REFERENCES

[1] S. Al-Khalifa et al. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] S. Amer-Yahia et al. TeXQuery: A Full-Text Search Extension to XQuery. In WWW 2004.

[3] S. Amer-Yahia et al. Tree pattern relaxation. In EDBT, 2002.

[4] K. Böhm et al. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. VLDB Journal Vol.6 No.4, Springer, 1997.

[5] J. M. Bremer and 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] N. Bruno et al. Top-K Selection Queries Over Relational Databases: Mapping Strategies and Performance Evaluation. ACM Transactions on Database Systems (TODS), 27(2), 2002.

[8] M. J. Carey and D. Kossmann. On Saying "Enough Already!" in SQL. In SIGMOD 1997.

[9] D. Carmel et al. Searching XML Documents via XML Fragments. In SIGIR 2003.

[10] C. Chen and Y. Ling. A Sampling-Based Estimator for Top-K Query. In ICDE 2002.

[11] T. T. Chinenyanga and N. Kushmerick. Expressive and Efficient Ranked Querying of XML Data. 4th International Workshop on the Web and Databases (WebDB). Santa Barbara, California, 2001.

[12] S. Cohen et al. XSEarch: A Semantic Search Engine for XML. In VLDB 2003.

[13] M. Cutler et al. Using the Structure of HTML Documents to Improve Retrieval. USENIX Symposium on Internet Technologies and Systems. California 1997.

[14] E. Damiani et al. The APPROXML Tool Demonstration. In EDBT 2002.

[15] 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).

[16] S. Flesca et al. On the minimization of XPath queries. In VLDB 2003: 153-164

[17] D. Florescu et al. Integrating Keyword Search into XML Query Processing. In WWW 2000.

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

[19] N. Fuhr, T. Rlleke. HySpirit a Probabilistic Inference Engine for Hypermedia Re- trieval in Large Databases. 6th International Conference on Extending Database Technology (EDBT). Valencia, Spain, 1998.

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

[21] Y. Hayashi et al. Searching Text-rich XML Documents with Relevance Ranking. ACM SIGIR 2000 Workshop on XML and Information Retrieval, Greece, 2000.

[22] V. Hristidis et al. PREFER: A system for the Efficient Execution Of Multiparametric Ranked Queries. In SIGMOD 2001.

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

[24] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In PODS 2002.

[25] S.-H. Myaeng et al. A Flexible Model for Retrieval of SGML Documents. ACM SIGIR Conference on Research and Development in Information Retrieval. Melbourne, Australia, 1998.

[26] J. Naughton et al. The Niagara Internet Query System. http://www.cs.wisc.edu/niagara/Publications.html

[27] N. Polyzotis et al. Selectivity Estimation for XML Twigs. ICDE 2004.

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

[29] A. Schmidt et al. Querying XML Documents Made Easy: Nearest Concept Queries. In ICDE 2001.

[30] T. Schlieder. Similarity Search in XML Data using Cost-Based Query Transformations. ACM SIGMOD 2001 Web and Databases Workshop. May, 2001. Santa Barbara, California.

[31] A. Theobald and G. Weikum. newblock Adding Relevance to XML. newblock 3rd International Workshop on the Web and Databases. Dallas, Texas, 2000.