Structure and Content Scoring for XML

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

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Amélie Marian
Columbia University
amelie@cs.columbia.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

David Toman
University of Waterloo
david@uwaterloo.ca

Abstract

XML repositories are usually queried both on structure and content. Due to structural heterogeneity of XML, queries are often interpreted approximately and their answers are returned ranked by scores. Computing answer scores in XML is an active area of research that oscillates between pure content scoring such as the well-known *tf*idf* and taking structure into account. However, none of the existing proposals *fully accounts for structure* and *combines it with content* to score query answers. We propose novel XML scoring methods that are *inspired by tf*idf* and that *account for both structure and content* while *considering query relaxations*. *Twig scoring*, accounts for *the most structure and content* and is thus used as our reference method. *Path scoring* is an approximation that *loosens* correlations between query nodes hence reducing the amount of time required to manipulate scores during top-k query processing. We propose efficient data structures in order to speed up ranked query processing. We run extensive experiments that validate our scoring methods and that show that path scoring provides very high precision while improving score computation time.

1 Introduction

XML data is now available in different forms ranging from persistent repositories such as the INEX and the US Library of Congress collections to streaming data such as stock quotes and news [16]. Such data is often queried on both structure and content [3, 6, 14, 18, 19]. Due to the structural heterogeneity of XML data, queries are usually interpreted approximately [1, 4, 5, 11, 15] and top-k answers are returned ranked by their relevance to the query. The term frequency (*tf*) and inverse document frequency (*idf*) measures, proposed in Information Retrieval (IR) [13], are widely used to score keyword queries, i.e., queries on content. However, although some recent proposals [3, 6, 11, 15] attempted to propose scoring methods that account for structure for ranking answers to XML queries, none of them fully captures *both structure and content* and uses *query relaxation* in computing answer scores.

In this paper, we propose scoring methods inspired by *tf*idf* to capture scoring and ranking queries both on structure and content. These methods rely on query relaxation techniques applied to structural predicates, i.e., XPath axes, such as in [1]. We define *twig scoring* as our method of reference as it accounts for all structural and content correlations in the query. However, it is time and space consuming because it requires computation of the scores of all relaxed versions of a query. Therefore, we propose *path scoring* as an approximation of twig scoring that *loosens* correlations between query nodes when computing scores, thereby reducing the amount of time required to compute and access scores during top-k query processing. The key idea in path scoring is to decompose the twig query into paths, compute the score of each path assuming independence between paths, and combine these scores into an answer score. This is in the same spirit as the vector space model of IR [13] where independence is assumed between query keywords and answer scores are computed as a combination of individual query keywords' scores.

In [10], we proposed *binary scoring* that also accounts for structural predicates and that computes answer scores by combining scores of individual child and descendants predicates in the query thereby assuming independence between all predicates. This scoring method is in fact an approximation of twig and path scoring that needs less time and space in exchange for a degradation in score quality.

Efficient top-k processing requires the ability to prune partial query matches, i.e., those that will never make the top-k answer list, as early as possible during query evalua-

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

tion. Given a query and a scoring method, different answers might have different scores depending on which relaxed form of the query they satisfy. In addition, the same answer might have a different score from one scoring method to another. However, all our scoring methods guarantee that *more precise answers to the user query are assigned higher scores*. This property can be used by any top-k algorithm since pruning is based on determining the most accurate score of a partial match using the query that the match satisfies best at a certain point in query evaluation and, identifying the best score growth of a partial match. Developing the right data structure and access method to store scores is a key factor in the efficient evaluation of ranked query answers [13]. We show how organizing query relaxations with their scores in a DAG structure, and using a matrix to quickly determine the score of a partial match, leads to efficient query evaluation and top-k processing.

To summarize, we make the following contributions:

- We propose *twig scoring*, a reference method for XML that is inspired by *tf*idf* in order to capture scoring query answers on both structure and content while accounting for query relaxation. We also propose *path scoring*, an approximation of *twig* scoring that reduces processing time. In addition, we discuss another approximation, *binary scoring*, that we previously proposed in [10]. All these scoring methods rely on the ability to evaluate structural predicates approximately.
- We propose a DAG to maintain precomputed *idf* scores for all possible relaxed queries that a partial match may satisfy. We use a matrix representation for queries, their relaxations, and partial matches to quickly determine the relaxed query that is best satisfied by a partial match during top-k query processing and prune irrelevant partial query matches.
- We implemented all our scoring methods in conjunction with a top-k processing algorithm. We ran extensive experiments on real and synthetic datasets and queries and showed that, compared to *twig scoring*, *path scoring* achieves very high precision for top-k queries while requiring moderate time; and that *binary scoring* results in high savings in time and space, but exhibits significant degradation in answer quality.

Related work is given in Section 2. Section 3 contains examples to motivate relaxation, scoring and top-k processing. Section 4 gives definitions and the implementation of our scoring. Experiments are detailed in Section 5. We conclude outlining several open issues in Section 6.

2 Related Work

Scoring for XML is an active area of research [1, 2, 4, 6, 7, 9, 10, 11, 14, 15, 17, 18, 19]. However, with the exception of [10], none of the existing proposals accounts for structural query relaxations while scoring on both structure and content. However, we show in this paper that the binary scoring method that we proposed in [10], while efficient, does not provide high quality answers compared to the reference twig scoring method.

The INitiative for the Evaluation of XML retrieval (INEX) [1] promotes new scoring methods for XML. INEX now provides a collection of documents as a testbed for various scoring methods in the same spirit as TREC was designed for keyword queries. Unfortunately, none of the proposed methods used in INEX as yet is based on structural relaxations to compute scores. As a result, the INEX datasets and queries would need to be extended to account for structural heterogeneity. Therefore, they could not be used to validate our scoring methods. As part of this effort, XIRQL [6] is based on a probabilistic approach [12] to compute scores at document edges and combines them to compute answer scores. The score of each keyword uses a path expression associated to the keyword in a query instead of document-based scores as in traditional IR [13]. However, no relaxations are applied to path expressions. Similarly, JuruXML [3] allows users to specify path expressions along with query keywords and modifies vector space scoring by incorporating a similarity measure based on the difference in length, referred to as length normalization, between the path expression provided in the query and the closest path in the data. We believe that relying on a principled way of applying relaxations to XPath queries carries more semantics than length normalization.

In [18], the authors study the relationship between scoring methods and XML indices for efficient ranking. They classify existing methods according to keyword and path axes. Based on that classification, they show that ranking on both structure and content are poorly supported by existing XML indices and propose IR-CADG, an extension to dataguides to account for keywords, that better integrates ranking on both structure and content. They show experimentally that this index outperforms existing indices that separate structure and content. This work is complementary to ours. It considers simple path queries and does not account for relaxations. It would be interesting to see how our DAG structure could be combined with the IR-CADG index to explore both structural relaxations and a tighter integration of indices on structure and on keywords.

Several query relaxation strategies for graph [8] and tree [4, 5, 6, 15] queries have been proposed before. In this paper, we adopt the relaxation framework defined in [1] since it captures most previously proposed relaxations and is general enough to incorporate new relaxations. While in [1], the focus was on defining a relaxation framework and query evaluation strategies assuming a given scoring function, in this paper, we focus on scoring methods and data structures to evaluate top-k XML queries.

3 Motivation

We represent XML data as forests of node labeled trees. Figure 1 shows a database instance containing fragments

[1] *http://inex.is.informatik.uni-duisburg.de:2004/*

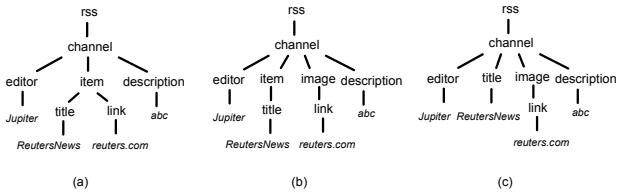

Figure 1: Heterogeneous XML Database Example

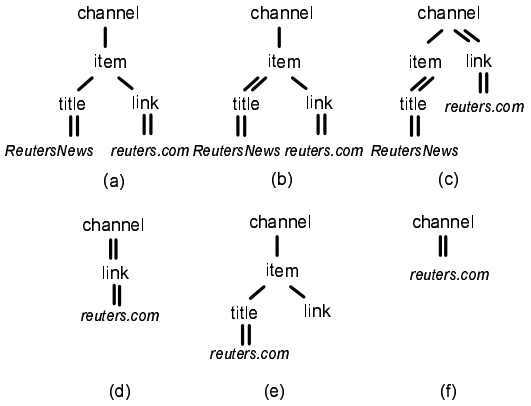

Figure 2: Query Tree Patterns and Relaxations

of heterogeneous news documents [16]. Figure 2 gives examples of several queries drawn as trees: the root nodes represent the returned answers, single and double edges the descendant and child axes, respectively, and node labels names of elements or keywords to be matched.

3.1 Query Relaxation

Different queries match different news documents in Figure 1. For example, query (a) in Figure 2 matches document (a) exactly, but would neither match document (b) (since `link` is not a child of `item`) nor document (c) (since `item` is entirely missing). Query (b) matches document (a) too since the only difference between this query and query (a) is the descendant axis between `item` and `title`. Query (c) matches both documents (a) and (b) since `link` is not required to be a child of `item` while query (d) matches all documents in Figure 1.

Intuitively, it makes sense to return all three news documents as candidate matches, suitably ranked based on their similarity to query (a) in Figure 2. Queries (b), (c) and (d) in Figure 2, correspond to *structural relaxations* of the initial query (a) as defined in [1].

In the same manner, none of the three documents in Figure 1 matches query (e) because none of their `title`s contains $reuters.com$. Query (f), on the other hand, is matched by all documents because the scope of $reuters.com$ is *broader* than in query (e). It is thus desirable to return these documents suitably ranked on their similarity to query (e).

In order to achieve the above goals, we use three relaxations: *edge generalization* (replacing a child axis with a descendant axis), *leaf deletion* (making a leaf node optional) and *subtree promotion* (moving a subtree from its parent node to its grand-parent). These relaxations capture all the structural and content approximations described in the examples. However, approximate keyword queries based on techniques such as stemming and ontologies [17], are orthogonal to and beyond the scope of this work.

Our relaxations capture approximate answers but still guarantee that exact matches to the original query continue to be matches to the relaxed query. For example, query (b) can be obtained from query (a) by applying edge relaxation to the axis between `item` and `title` and still guarantees that documents where `title` is a child of `item` are matched. Query (c) is obtained from query (a) by composing edge generalization between `item` and `title` and subtree promotion (applied to the subtree rooted at `link`). Finally, query (d) is obtained from query (c) by applying leaf deletion to the nodes $ReutersNews$, `title` and `item`. Query (d) is a relaxation of query (c) which is a relaxation of query (b) which is a relaxation of query (a). Similarly, query (f) in Figure 2 can be obtained from query (e) by a combination of subtree promotion and leaf deletion.

3.2 Answer Scoring

In order to distinguish between answers to different relaxations of the same query, we need a scoring method to compute the relevance of each query answer to the initial user query. The traditional *tf***idf* measure is defined in IR for keyword queries against a document collection. The *idf*, or inverse document frequency, quantifies the relative importance of an individual keyword in the collection of documents. The *tf*, or term frequency, quantifies the relative importance of a keyword in an individual document. In the vector space model [13], query keywords are assumed to be independent of each other, and the *tf***idf* contribution of each keyword is added to compute the final score of a document.

In our context, the most *accurate* scoring method would compute the score of an answer taking occurrences of *all* structural and content (i.e., keyword) predicates in the query. For example, a match to query (c) would be assigned an *idf* score based on the fraction of the number of `channel` nodes that have a child `item` with a descendant `title` containing the keyword $ReutersNews$ and a descendant `link` that contains the keyword $reuters.com$. Such a match would be assigned a *tf* score based on the number of query matches for the specific `channel` answer. We refer to this method as *twig scoring*.

While twig scoring captures all correlations between nodes in the query, it is time and memory consuming because it requires to compute the scores of each relaxed query. Therefore, we define *path scoring* that *loosens* the correlations between query nodes by assuming independence between root-to-leaf paths in the query, computing their scores and combining those scores to compute an answer score. For example, for query (a) in Figure 2, twig scoring is based on the number of `channel` nodes that have an `item` with a `title` containing $ReutersNews$ and a

`link` containing $reuters.com$ while path scoring relies on decomposing the query into its two paths, computing their scores separately and combining them to computer an answer score. Hence, it might not always *distinguish between answers to different relaxed queries* as well as twig scoring.

The scoring method proposed in [10] is another approximation of twig scoring. We refer to it as *binary scoring* because it scores binary predicates with respect to the query root, and assumes *independence* between those predicates. In query (a) in Figure 2, that would amount to computing the scores of the child predicate between `channel` and `item` and the scores of descendant predicates between `channel` and each one of the remaining nodes in the query including keyword nodes.

3.3 Top-k Processing

Scores need to be organized in such a way that helps to determine the highest score of a partial match during top-k processing in order to speed up pruning of irrelevant answers. To avoid computing scores on-demand, query evaluation could take advantage of the fact that *idf*s are shared across all partial matches that satisfy the same (relaxed) query. For example, all answers that match query (b) in Figure 2 and not query (a) would have the same *idf*. Therefore, we propose to precompute and store *idf*s for all possible relaxations of the user query. This allows for fast access to score values during query processing. We use two data structures: a *Query Relaxations DAG*, and a *Query Matrix*, discussed in the next section.

4 Scoring

In this section, we formally define approximate answers to twig queries based on the notion of query relaxation and the corresponding scoring methods.

4.1 Twig Queries and Relaxations

We use previously defined *twig queries*, an important subset of XPath. A twig query Q (on k nodes) is a rooted tree with string-labeled nodes and two types of edges, $/$ (a child edge) and $//$ (a descendant edge).[2] We call the root node root_Q of Q the *distinguished answer node*.

We use the term *match* to denote the assignments of query nodes to document nodes that satisfy the constraints imposed by the query and the term *answer* to denote document nodes for which there is a match that maps the root of the query to such a node. Note that for a particular answer there can be multiple matches in a document. For example, in the document "$< a >< b/ >< b/ >< /a >$" there are two matches but only one answer to the query a/b. We denote $Q(D)$ the set of all answers to Q in a document D.

[2]Our scoring methods can be defined for twig queries with any XPath axis edges. Edge generalization and composition (subtree promotion) of edges can be defined for all XPath axes, e.g., `a/parent::b` can be generalized to `a/ancestor::b`, `a/child::b/following-sibling::c` can have the subtree promotion `a[./child::b]/child::c`. For simplicity of exposition, we do not investigate this issue further in this paper.

Definition 1 *Let Q and Q' be twig queries. We say that Q'* subsumes *Q if $Q(D) \subseteq Q'(D)$ for all documents D.*

To capture approximate answers to a given twig query we generate relaxed twig queries on a subset of the query nodes based on the following notion of *query relaxation*:

Definition 2 (Relaxation) *Let Q be a twig query. We say that Q' is a* simple relaxation *of Q (and write $Q \mapsto Q'$) if Q' has been obtained from Q in one of the following ways:*

- *an* edge generalization *relaxation: a $/$ edge in Q is replaced by a $//$ edge to obtain Q';*
- *a* subtree promotion *relaxation: a pattern $a[b[Q1]//Q2]$ is replaced by $a[b[Q1]and.//Q2]$; or*
- *a* leaf node deletion *relaxation: a pattern $a[Q1and.//b]$ where a is the root of the query and b is a leaf node is replaced by $a[Q1]$.*

We say that Q' is a relaxation *of Q (and write $Q \mapsto^* Q'$) if it is obtained from Q by a composition of k simple relaxations ($k \geq 0$).*

Note that, given a query Q with the root labeled by a, the *most general* relaxation is the query a. We denote this query by $Q_\perp$. Every exact answer to a relaxation of Q is an approximate answer to Q, and the set of all *approximate answers to* Q in a document D is equal to $Q_\perp(D)$.

The relaxations defined above do not capture approximating content such as using stemming or ontologies on keywords [17]. While a detailed discussion of this direction is beyond the scope of the paper, the actual way of relaxing matches to keywords is orthogonal to the remaining development in the paper.

We organize the set of all relaxations of a query into a directed acyclic graph (DAG) in which edges relate relaxations in a subsumption relation. We need two preliminary lemmas:

Lemma 3 *Let Q and Q' be twig queries such that $Q \mapsto^* Q'$. Then $Q(D) \subseteq Q'(D)$ for all documents D.*

Proof: Each simple relaxation satisfies the statement of the lemma (by inspection); the rest follows from transitivity of the inclusion relation. □

Lemma 4 *Let Q and Q' be two twig queries such that $Q \mapsto^* Q'$ and $Q' \mapsto^* Q$. Then $Q = Q'$.*

Proof: From Lemma 3 we know that $Q(D) \subseteq Q'(D)$ and $Q'(D) \subseteq Q(D)$ for all documents D. Thus $Q \equiv Q'$. However, this is only possible if $Q = Q'$ (syntactically) as each simple relaxation produces a strictly less restrictive query. □

Equipped with these two lemmas we can organize the relaxations in a DAG as follows:

Definition 5 (Relaxation DAG) *Let Q be a twig query. We define*

$$\text{RelDAG}_Q = (\{Q' | Q \mapsto^* Q'\}, \{(Q', Q'') | Q \mapsto^* Q' \wedge Q' \mapsto Q''\})$$

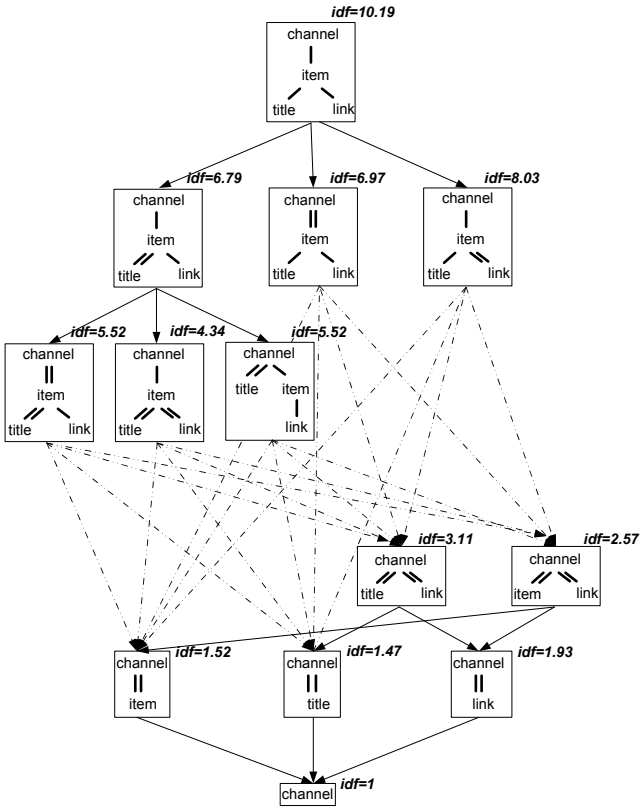

Figure 3: A Query Relaxations DAG

Figure 3, disregarding the numerical scores attached to nodes, shows the DAG created for a simplified query (a) in Figure 2. Given a query Q, Algorithm 1 is used to build the DAG in a top-down fashion, starting with a node containing the query Q, applying the simple relaxation steps, and merging identical DAG nodes on the fly. This leads to the following result:

Algorithm 1 buildDAG Function

```
Require: currentDAGNode
 1: Q=getQuery(currentDAGNode);
 2: for each node n in Q do
 3:    if (canBeRelaxed(n,parent(n))) then
 4:       newDAGNode = getDAGNode(edgeGeneralize(n,Q));
          {getDAGNode checks if a DAG node containing query Q with
          the edge generalized exists, returns it if it does or creates it if it
          does not.}
 5:    else if (not isQueryRoot(parent(n))) then
 6:       newDAGNode = getDagNode(promoteSubtree(n,Q));
 7:    else if noDescendants(n) then
 8:       newDAGNode = getDAGNode(leafDelete(n,Q));
 9:    end if
10:    addChild(currentDAGNode,newDAGNode); {addChild adds
       newDAGNode as a child of currentDAGNode in the DAG.}
11:    buildDAG(newDAGNode); {recursive call on newDAGNode.}
12: end for
```

Theorem 6 *Let Q be a twig query. Then Algorithm 1 produces* RelDAG_Q.

Proof: For a query relaxation node in the DAG and for every node in that relaxation the algorithm applies all allowed simple relaxations (cf. Definition 2: only one simple relaxation applies per node in a query). Nodes representing these relaxations become the children of this node in the DAG; new nodes are only created when they don't already exist. The reminder of the proof is a simple induction on the distance of a node from the root of the DAG. Termination of the algorithm is guaranteed as there are only finitely many relaxations of a given query. □

4.2 Scoring Twig Answers

As the *approximate* answers to a query Q are simply answers to the relaxation $Q_\perp$, our goal in this section is to rank elements of $Q_\perp(D)$ by assigning numerical values using a *scoring function*. The basic idea is that scores are based on considering best matches for a given answer—matches to the least relaxed query in the DAG.

We base our scoring function on the *tf*idf* measure proposed in IR [13]. However, we have to modify this measure to distinguish among matches to different relaxations of the original query. We first define the modification of the *inverse document frequency* (*idf*):

Definition 7 (*idf* of a Relaxation) *Let Q and Q' be twig queries such that $Q \mapsto^* Q'$ and D an XML document. We define*

$$\mathrm{IDF}_D^Q(Q') = |Q_\perp(D)|/|Q'(D)|;$$

We extend this measure to all approximate answers $e \in Q_\perp$ by defining

$$\mathrm{IDF}_D^Q(e) = \max\{\mathrm{IDF}_D^Q(Q') | e \in Q'(D), Q \mapsto^* Q'\}.$$

We say that a relaxation Q' that maximizes $\mathrm{IDF}_D^Q(e)$ is a most specific relaxation of Q for e *and denote the set of these relaxations by $\mathsf{MSR}_D^Q(e)$.*

Our *idf* scoring guarantees that answers to less approximate queries obtain *idf* scores at least as high as scores to more approximate ones; this is also the basis for assuring that the *score-monotonicity* requirement is met by the overall score of an answer. In particular:

Lemma 8 *Let Q' and Q'' be two relaxations of Q such that $Q' \mapsto^* Q''$. Then $\mathrm{IDF}_D^Q(Q') \geq \mathrm{IDF}_D^Q(Q'')$ for any document D.*

Proof: By definition, any answer to Q'' is an answer to Q', as Q'' is a relaxed version of Q'. Therefore, the denominator value in the *idf* function for the computation of Q'' is greater than or equal to the denominator value in the *idf* function for the computation of Q. It results that $\mathrm{IDF}_D^Q(Q') \geq \mathrm{IDF}_D^Q(Q'')$. □

Thus, since the *idf* score for an answer e is defined as the maximal *idf* value of all relaxations of Q having e as an answer, the above lemma also shows that $\mathrm{IDF}_D^Q(e) \geq \mathrm{IDF}_D^Q(e')$ whenever the best match for e matches a less relaxed query than the best match for e'.

Intuitively, the *idf* measure of a query Q quantifies the extent to which answers to $Q_\perp$ in D additionally satisfy Q. Thus, more selective queries are assigned higher *idf* scores. This is akin to the IR case: keywords that appear in a document collection less frequently are assigned higher *idf* scores.

Note, however, that the *idf* measure defined above assigns the same *idf* score to all exact matches to a query Q. In general, all answers having their best match with respect to the same relaxed query are given the same *idf* score. On the other hand, the *idf* measure becomes useful once we allow for relaxed matches to the query Q, as described in Section 3. The *idf* scores are then used to rank relaxed matches based on how closely they match the relaxed query. To distinguish between matches of the same relaxed query we use the analogue of the *term frequency* (*tf*) measure:

Definition 9 (*tf* of an Answer) *Let Q be a twig query and D an XML document. Then, for an answer $e \in Q(D)$, we define*

$$\mathrm{TF}_D^Q(e, Q') = |\{f | f \text{ a match of } Q' \text{ in } D, f(\mathrm{root}_{Q'}) = e\}|$$

for Q' a most specific relaxation of Q for e, and

$$\mathrm{TF}_D^Q(e) = \max\{\mathrm{TF}_D^Q(e, Q') | Q' \in \mathsf{MSR}_D^Q(e)\}.$$

Intuitively, the *tf* score of an answer quantifies the number of distinct ways in which an answer matches a query. This is again akin to the IR case where the term frequency increases with the number of occurrences of a keyword in a document. The final scoring function for twig queries is based on combining the *idf* and *tf* scores. We use a *lexicographical* (*idf*, *tf*) ordering to satisfy the *score monotonicity* requirement.

Definition 10 (Lexicographical Score) *Let D be an XML document, Q a query, and e and e' approximate answers to Q in D. We define*

$$e \leq e' \text{ if } (\mathrm{IDF}_D^Q(e) < \mathrm{IDF}_D^Q(e')) \text{ or } (\mathrm{IDF}_D^Q(e) = \mathrm{IDF}_D^Q(e') \text{ and } \mathrm{TF}_D^Q(e) \leq \mathrm{TF}_D^Q(e'))$$

Using this definition and Lemma 8 we have:

Theorem 11 *Let $e, e' \in Q_\perp(D)$, Q' and Q'' be the most specific relaxations of Q for e and e' in D, respectively, such that $Q' \mapsto^* Q''$. Then $e' \leq e$.*

Note that the more common combinations of the *tf* and *idf* scores, e.g., the $\mathrm{TF}_D^Q(e) * \mathrm{IDF}_D^Q(e)$ function, do not adhere to our requirement of matches to less relaxed queries to be ranked higher. Consider, for example, the query a/b posed over the concatenation of two documents "$< a >< b/ >< /a >$" and "$< a >< c >< b/ > ... < /c >< a >$" with $\ell > 2$ nested "b" elements. Then the *idf* scores for a/b and the relaxation $a//b$ are 2 and 1, respectively. However, the *tf* measures are 1 and ℓ. Thus the more common *tf*idf* ranking would prefer the second (less precise) answer. Note also, that dampening the *tf* factor, e.g., using a log function, cannot solve this *inversion* problem as we can choose ℓ to be arbitrarily large.

4.3 Scoring for Path/Binary Approximations

We use twig scoring as the *reference* measure of correctness since it accounts for the most structure and content. However, to compute the scores of answers we need to have access to the *idf* scores associated to all relaxations of the original query. As we pointed out in the previous section, computing (or even precomputing whenever possible) these scores can be very expensive. Thus in order to improve efficiency of the overall query processing, we define approaches based on decomposing an original twig query to simpler queries and this way we reduce the number of different *idf* scores needed. Also, in many cases the scores for such simpler queries are easier to compute. In particular, we consider two decompositions $\mathsf{decomp}(Q)$ for a twig query Q:

Path Decomposition the set of all paths in Q leading from the root of Q to any other node in Q; and

Binary Decomposition the set of all queries $Q_i = \mathrm{root}_Q/m$ or $Q_i = \mathrm{root}_Q//m$ for m a node in Q such that $Q \subseteq Q_i$.

The decompositions for our example query are as follows:

Example 12 *The sets of queries*

$$\{\mathrm{channel/item/title}, \mathrm{channel/item/link}\},$$
$$\{\mathrm{channel/item}, \mathrm{channel//title}, \mathrm{channel//link}\}$$

are the Path *and* Binary *Decompositions, respectively, of a twig query "*$\mathrm{channel/item[./title]/link}$*".*

For each decomposition, we also need to define how the scores for the individual fragments are combined into a final answer score. The *idf* measure depends on whether we consider joint (*correlated*) matches only or assume *independence* between matches to the individual components of a twig query. Hence, we have two definitions of *idf*: one for the correlated case and one for the independent case.

Definition 13 (Path/Binary *idf* Score) *Let Q be a twig query, Q' a relaxation of Q, and D an XML document. We define*

$$\mathrm{IDF}_D^Q(Q') = |Q_\perp(D)|/|\bigcap_{Q_i \in \mathsf{decomp}(Q')} Q_i(D)|$$

for correlated scoring, *and*

$$\mathrm{IDF}_D^Q(Q') = \sum_{Q_i \in \mathsf{decomp}(Q')} |Q_\perp(D)|/|Q_i(D)|$$

for independent scoring.

The *idf* score of an answer under the above assumptions is again the maximal *idf* of a relaxation containing the answer.

The *tf* measure is the same in both cases as it is defined on a per-answer basis:

Definition 14 (*tf* for Path/Binary) *Let Q be a twig query, Q' a relaxation of Q and D an XML document. Then, for $e \in Q(D)$, we define*

$$\mathrm{TF}_D^Q(e, Q') = \prod_{Q_i \in \mathsf{decomp}(Q')} \left| \left\{ f | f \text{ a match of } Q_i \text{ in } D, \atop f(\mathrm{root}_{Q_i}) = e \right\} \right|$$

where Q' is a most specific relaxation for e, and

$$\mathrm{TF}_D^Q(e) = \max\{\mathrm{TF}_D^Q(e, Q') | Q' \in \mathsf{MSR}_D^Q(e)\}.$$

Similarly to the twig scoring, we can show that the lexicographical (*idf*, *tf*) ordering of query answers based on the scores obeys the score monotonicity requirement.

Note that the distinction between *independent* and *correlated* scoring only applies for *binary* and *path* scoring. Altogether we have defined five scoring methods, listed in the order of increasing precision: *binary-independent* that considers all predicates to be independent, *binary-correlated* that takes into account correlations between individual binary predicates, *path-independent* that assumes independence between query paths, *path-correlated*, that takes into account the correlation both within paths and across paths in the query, and *twig*, the reference scoring method, that takes all of the query twig correlations into account.

4.4 Top-k Query Processing

In this section, we discuss data structures that can be used by *any* top-k processing algorithm to compute top-k answers to XML queries efficiently.

4.4.1 Using the DAG

As mentioned earlier, our DAG provides a convenient, constant-time access to the *idf* value of any partial match during query processing (see Figure 3 for an example). This value can be computed using selectivity estimation techniques for twig queries [11].

Note that every (even partial) match is an exact match to a relaxation of the original query. Also, for matches we have:

Lemma 15 *Let Q be a query, D an XML document, and f a match for an answer $e \in Q_\perp(D)$. Then there is a unique query $Q' \in \mathrm{RelDAG}_Q$ such that f is a match for $e \in Q'(D)$ and f is not a match for e in $Q''(D)$ for any ancestor Q'' of Q' in RelDAG_Q.*

Thus it is sufficient to associate a single score with every match. At each DAG node, we keep the maximum theoretical upper bound for a partial match that satisfies the twig query associated with that node: if the query at that node includes all nodes of the original query, then a partial match that satisfies this twig query cannot be further extended, and its score upper bound value is equal to its *idf* value; however, if the twig query does not include all the nodes from the original query, e.g., if it is a relaxation of the original query where some leaf deletion operations were applied, we store a pointer in the DAG to the DAG node containing the best relaxation such an incomplete partial match could satisfy. In the same manner, we can keep pointers in the DAG to access information such as the score upper bound values of all possible configurations of partial matches (some nodes missing, some nodes unknown), or the maximum score increase (in *idf* value) that would be gained from checking one of possible unknown nodes in the partial match. During query evaluation, *idf*s are accessed in constant time using a hash table to check the query partial matches against the twig queries stored in the DAG.

From Lemma 8, it follows that the deeper a query is in the DAG, the lower its *idf* is. An example of a query relaxation DAG for the (simplified) query from Figure 2(a) is given in Figure 3. Note that a, the lowest (most relaxed) query in the DAG, has an *idf* of 1 as it consists of returning every single distinguished node.

4.4.2 Using the Matrix

We propose a query matrix used to apply relaxations to queries during the DAG building step and, more importantly, to map a partial match to its corresponding query using matrix subsumption during query evaluation. By representing both partial matches and queries in the same framework, we can compare them efficiently, by only requiring a matrix comparison.

The matrix is defined for twig queries on m nodes; we assume that the nodes are named $\{n_1, \ldots n_m\}$.

Definition 16 (Matrix Representation) *Let Q be a twig query on at most m nodes. We define a $m \times m$ matrix M_Q as follows:*

- *$M[i,i] = a$ if the node $n_i \in Q$ has label a;*
  $M[i,i] = X$ if the node $n_i \notin Q$;
  $M[i,i] = ?$ otherwise;
- *$M[i,j] = /$ if $n_i/n_j \in Q$;*
  $M[i,j] = //$ if there is a path from n_i to n_j in Q but $n_i/n_j \notin Q$;
  $M[i,j] = X$ if $n_i, n_j \in Q$ and there is no path from n_i to n_j;
  $M[i,j] = ?$ otherwise.

A subsumption order *between the symbols stored in the matrix cells is defined as follows: $a < ?$, $/ < // < ?$, and $X < ?$. The reflexive subsumption order $\leq$ is the above order extended with the diagonal relation on the symbols. A partial match matrix can be defined similarly.*

It is easy to see that a lower matrix is sufficient to capture all the information represented in Q as queries are trees.

Figure 4 shows the query matrix 4(a) for the (simplified) query from Figure 2(a), and several possible partial matches to this query that can be computed during query evaluation: 4(b) is a partial match that has not yet been

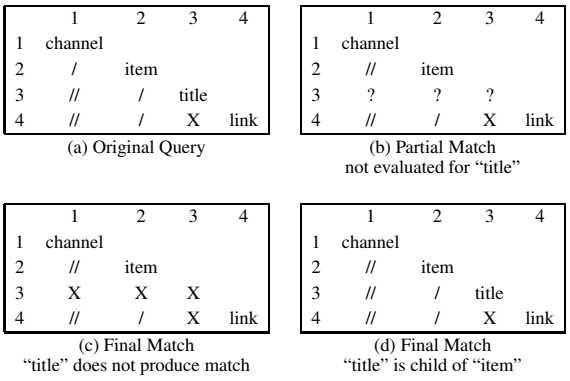

Figure 4: Query Matrices

checked against `title`, hence the corresponding entries are set to "?"; the relationship between `channel` and `item` for that partial match has been relaxed to "//". 4(c) is the same partial match as 4(b), with no `title` nodes found, the corresponding matrix entries are then equal to "X". Finally, 4(d) is a extension of 4(b) for which an exact match for node `title` has been found.

Matrices are created for partial matches by checking their binary node relationships. Operations on matrices are performed in three situations: to create a relaxed version of a query in the DAG building process (e.g., by replacing all entries involved in an edge generalization with their relaxation), to check whether a query is a relaxation of another query (matrix subsumption), or to check whether a partial match maps to a query pattern (matrix subsumption).

Matrix creation and subsumption operations need $(m^2/2)$ comparison where m is the number of query nodes. Since queries are expected to be fairly small, most often no larger than 10 nodes, this produces efficient computation times. Each matrix entry has a maximum of 4 possible entries therefore there are at most $4^{m^2/2}$ relaxations of a given query; the actual number tends to be much lower as most matrix combinations are not possible. This loose upper bound also gives us an upper bound on the size of the DAG, as there is only one DAG node per query relaxation.

4.4.3 Top-k Algorithm

In this paper, we do not claim the top-k processing algorithm as a contribution since we use the adaptive processing algorithm developed in [10]. However, our DAG and matrix data structures could be used by any top-k algorithm to determine (i) the highest score of a partial match during query evaluation and (ii) if a partial match should be pruned or not depending on its score upper bound.

Algorithm 2 is a sketch of the top-k algorithm that we use. It starts by evaluating the query root node. Then, it determines the partial matches with the highest score potential using `getHighestPotential` which relies on score upper bounds extracted from the DAG to prioritize partial matches. The algorithm then expands those matches by computing the next best query node for each one of them. Note that the algorithm treats each partial match individually (as opposed to a batch processing). When a partial match is generated, it is checked against the top-k list (`updateTopK`). The partial match may be used to update the top-k list or it may be carried to the next step or it may be pruned. The algorithm stops when all query nodes have been evaluated for all matches in the top-k list and there is no other match that is waiting to be processed.

Algorithm 2 A Generic top-k Algorithm

Require: Query Q, Document D
1: PartialMatches=getRootNodes(D,Q);
{PartialMatches is the set of partially evaluated answers.}
2: topK=empty;
{the while condition checks if all K answers in topK are complete, and if no partial match has potential final scores higher than current topK matches.}
3: **while** !checkTopK(topK) **do**
4: currentMatch=getHighestPotential(PartialMatches);
{partial match with the highest potential final score.}
5: newMatches=expandMatch(currentMatch,Q,D);
{chooses the next best query node to evaluate for currentMatch and may generate many new matches.}
6: PartialMatches+=newMatches;
{adds expanded partial matches to PartialMatches.}
7: topK=updateTopK(newMatches);
{updateTopK keeps the best K answers in topK}
8: **end while**

5 Experimental Evaluation

In this section we briefly discuss our implementation of top-k query processing techniques and then present extensive quality and efficiency evaluations of the proposed XML scoring methods.

5.1 Summary of Results

Our experimental evaluation compares the five scoring methods: *binary-independent*, *binary-correlated*, *path-independent*, *path-correlated*, and *twig*. *Twig* results in the perfect top-k answer. Our results show that the *binary* scoring methods allow for fast DAG preprocessing and query execution times, in exchange for degraded answer quality. When score quality is important, both *path* methods offer good quality answers, but *path-correlated* requires high preprocessing times. In contrast, *path-independent* offers good answer quality (often perfect), while saving in terms of preprocessing times.

We implemented our top-k strategies such that all *idf*s and score upper bounds are accessed through the DAG. Our *(idf,tf)* scoring measure (see Section 4.2) assigns the same *idf*s to matches that share the same (relaxed) query pattern. Ties on such matches are broken based on the answers *tf*s. Since, unlike *idf*, each match has an individual *tf* score, it is more efficient to estimate the *tf* of a match during query evaluation based on selectivity estimates (which can be stored in the DAG). However, in order to avoid skewing results in our experimental evaluation of *idf* scoring, we do not take *tf*s into account.

5.2 Experimental Setup

We implemented the DAG and query matrix structures, as well as the top-k query processing strategies from [10] in C++. We ran our experiments on a Red Hat 7.1 Linux 1.4GHz dual-processor machine with 2Gb of RAM.

5.2.1 Data and Queries

To offer a comprehensive evaluation of our scoring methods, in terms of time and space, as well as their effect on query processing, we performed experiments on synthetic XML data. Results on real data are given in Section 5.3.5.

We generated heterogeneous collections of documents using the Toxgene document generating tool[3]. In order to enable query relaxation, documents of various sizes were generated using heterogeneous DTDs. For our synthetic experiments, the created documents contain simple node labels (e.g., <a> and <b>), and U.S. state names as text content. We then ran our experiments on different datasets by assembling documents based on size (in terms of number of nodes). We also performed experiments on collections where we varied the parameters of the datasets such as correlation or number of exact answers. We measured the correlation of a dataset as the type of matches to query predicates that are present in the dataset: simple binary predicates (no correlation), binary predicates only, binary predicates and simple path predicates, binary and path predicates, and mixed (all three types of predicates are present in the dataset). The number of exact answers is a percentage of the top-k answers that are exact answers to the query. We report our values for correlation and the number of exact answers with respect to our default query $q3$.

We evaluated our scoring methods on 18 different queries exhibiting different sizes, query structures (twig shapes), and content predicates. We chose these 18 queries to illustrate the different possible query relaxation structures that may happen in a real-world scenario.

$q0$: `a[./b/c]`
$q1$: `a[./b][./c]`
$q2$: `a[./b/c/d]`
$q3$: `a[./b[./c/]/d]`
$q4$: `a[./b][./c][./d]`
$q5$: `a[./b/c/d/e]`
$q6$: `a[./b[./c]/d/e]`
$q7$: `a[./b/c/d/e/f]`
$q8$: `a[./b[./c/d]/e/f]`
$q9$: `a[./b[./c[./e]/f]/d][./g]`
$q10$: `a[contains(./b,"AZ")]`
$q11$: `a[contains(.,"WI") and contains(.,"CA")]`
$q12$: `a[contains(./b/c,"AL")]`
$q13$: `a[contains(./b,"AL") and contains(./b,"AZ")]`
$q14$:
```
a[contains(.,"WA") and contains(.,"NV") and
    contains(.,"AR")]
```
$q15$:
```
a[contains(./b,"NY") and
    contains(./b/d,"NJ")]
```
$q16$: `a[contains(./b/c/d/e,"TX")]`
$q17$:
```
a[contains(./b/c,"TX") and
    contains(./b/e,"VT")]
```

[3] *http://www.cs.toronto.edu/tox/toxgene/*

We performed our synthetic data experiments varying different parameters: query size, query shape, document size (in terms of number of nodes that satisfy each query node), document correlation, number of exact answers, k. The default parameters we used for our experiments are summarized in Table 1.

Finally, we also ran several experiments on a real dataset: the XML version of the Wall Street Journal Treebank[4] corpora. Treebank provides text annotations of English sentences, the dataset we use consists of annotated Wall Street Journal text. Sentences are broken using tags representing various grammatical (phrases) and speech structures. For instance, <NP> represents a noun phrase within a sentence (<S>), the noun phrase can include different part-of-speech such as a singular noun (<NN>). Tags used in the queries we tested include: prepositional phrase (<PP>), verb phrase (<VP>), determiner (<DT>), interjection, (<UH>), comparative adverb (<RBR>), and possessive ending (<POS>). We ran experiments on 6 queries of different sizes and shapes:

TB0:
```
S[./UH and contains(./VP,"There")]
```
TB1: `S[./NN[./NP]/DT]`
TB2: `NP[./S/PP/NN]`
TB3: `VP[./S/[./NP]/PP/NN]`
TB4: `VP[./S/NP/PP/NN]`
TB5:
```
S[./VP[./RBR][./POS] and
    contains(./VP,"should")]
```

5.2.2 Evaluation Measures

To compare the performance of the *idf* scoring mechanisms, we used the following measures:

DAG Size: Memory size needed to store the DAG structure. This shows the memory size needed for each method.
DAG Preprocessing Time: Time needed to build the DAG, compute the *idf* scores and all optional information stored in the DAG. In order to isolate the effect on scores approximation due to binary and path scoring methods, we computed the exact *idf* scores by exploring all matches. This preprocessing step can be improved using selectivity estimation methods such as in [11].
Precision: Percentage of top-k answers (and their ties) that are correct top-k answers (or ties to the correct top-k answer), according to the exact *twig* scoring method. Answer ties are answers to the query that share the same *idf* as the k^{th} returned answer. Our Precision measure takes possible ties into account in order to penalize scoring methods that produce too many possible top-k results (i.e., scoring methods that produce many answers with the same score) compared to the *twig* method. The precision measure gives some information about the quality of the answers returned.
Query Processing Time: Time needed to compute the top-k answer to the query, in addition to the DAG preprocessing time. This measure shows how score distribution impacts query processing time.

[4] *http://www.cis.upenn.edu/ treebank/home.html*
http://www.cs.washington.edu/research/xmldatasets/www/repository.html

Query Size	Query Shape	Document Size	Document Correlation	# of Exact Answers	k
$q3$ (4 nodes)	$q3$ (twig)	$[0, 1000]$	Mixed (with respect to $q3$)	12% (with respect to $q3$)	25

Table 1: Experimental Default Settings

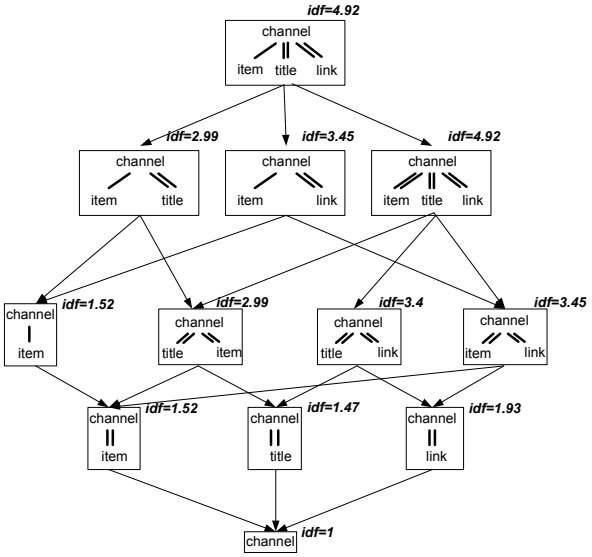

Figure 5: A Query Relaxations DAG for *binary Scoring*

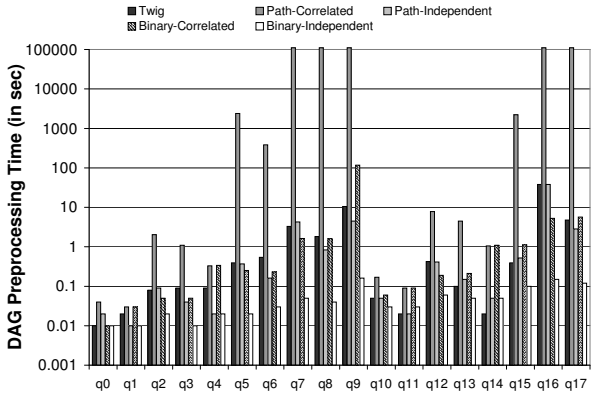

Figure 6: DAG Preprocessing Time

5.3 Experimental Results

We now present our experimental results for the evaluation of our different scoring strategies.

5.3.1 DAG Size

The *path* and *twig* scoring potentially result in different *idf* score values for each node in the relaxation DAG described in Section 4. *Binary* scoring does not assign different *idf*s to all DAG nodes, but only to those that result in different binary query structures. In order to save memory space, and DAG preprocessing time, it is therefore possible to only build a subset of the relaxation DAG when considering *binary* scores. A simple way to implement this optimization is to convert the original query into a binary predicate query, and build the relaxation DAG from this transformed query. Figure 5 shows the DAG that results from binary scoring (assuming independent predicate scoring for the *idf* scores) of the query in Figure 3. Since the binary version of the query is much simpler than the query itself and results in fewer possible relaxations, its DAG is smaller than, or the same size as the original relaxation DAG; 12 nodes vs. 36 nodes in our example. Experimental evaluation shows that for queries that do not only consist of binary predicates but also offer some complex structural patterns, the DAGs for the *twig* and *path* scoring methods are an order of magnitude larger than the DAGs for the *binary* scoring methods. However, these more complex DAGs are still of a reasonable size (1MB for our larger query $q9$), and can therefore be easily kept in main memory during top-k query processing.

5.3.2 Comparing Scoring Methods

The preprocessing times needed to build the DAGs and compute the *idf* scores for each of our scoring methods are shown, for all 18 queries over a small dataset, in Figure 6 (logarithmic scale). The *path-correlated* method is the most expensive and its cost grows rapidly with the query size, with times greater than 100,000 secs for $q7$, $q8$, $q9$, $q16$ and $q17$; its high cost is mostly due to very expensive score computation and propagation. *Path-independent* and *twig* are faster, with *path-independent* faster than *twig* for all non-chain queries. For chain queries: $q0$, $q2$, $q5$, $q7$, $q10$, $q12$ and $q16$, the preprocessing times of *twig* and *path-independent* are similar. For these two methods, exploring all matches dominates preprocessing time. In the case of chain queries, the matches considered are the same for *twig* and *path-independent*, *path-independent* is slightly slower due to some score propagation (sum computation overhead). Note that this overhead will become negligible as the document collection size increases. The two *binary* methods are faster than their *path* counterparts, as they work on a smaller DAG, but they offer smaller score ranges. *Binary-correlated* can be expensive as the query size increases, and is often more expensive than *twig*. Since both *correlated* methods are outperformed by *twig*, we will not report further results for these methods in this paper.

Figure 7 shows the precision of top-k query evaluation strategies when using the three remaining methods. The *twig* method has the perfect precision. *Path-independent* has very good precision, often equal to 1, or close. *binary-independent* has the worst precision, as it does not offer a fine granularity of scores. If time is the main constraint, then *binary-independent* allows for fast preprocessing time in exchange for some degradation in score quality. If score quality is important, for chain queries, the *twig* approach is the best as it provides perfect precision, and is as fast as *path-independent*; for queries having more

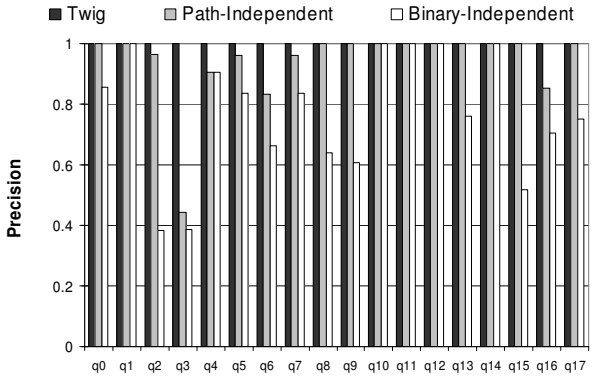

Figure 7: Top-k Precision

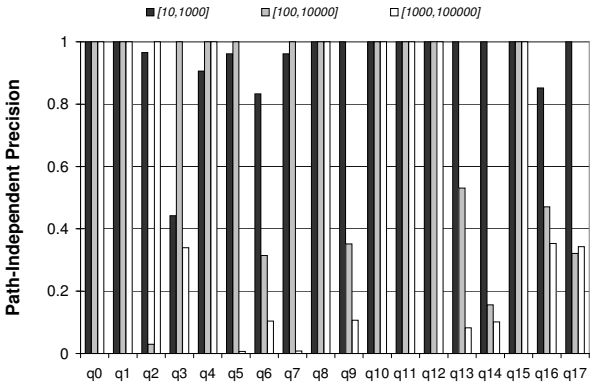

Figure 8: Top-k Precision for *path-Independent* Scoring

complex shapes, *path-independent* provides the best quality/preprocessing time tradeoff.

We also compared the processing time needed to evaluate a top-k query with the different scoring methods using the top-k query evaluation strategies from [10]. The *twig* and *path* techniques results in similar query execution times. However, we observed that the *binary* approaches may result in slightly faster query processing times, as more partial matches end up with the highest scores, allowing to identify a top-k set earlier in the execution and discard low-quality matches faster. This makes *binary-independent* the method of choice when time is an issue. An in-depth comparison of the performance of top-k query processing strategies is beyond the scope of this paper. We refer the reader to [10] for more details on this subject.

In the rest of this section, we study the different parameters that affect quality and speed of our proposed scoring methods.

5.3.3 Varying the Document Collection Size

Figures 8 shows the effect of document size, in terms of the number of document nodes that match each query node, on the precision of *path-independent* on a subset of the synthetic data queries. While precision is mostly affected by data distribution, larger documents may end up producing more ties to the top-k answers, which in turn leads to lower precision values. Precision results for *path-independent*

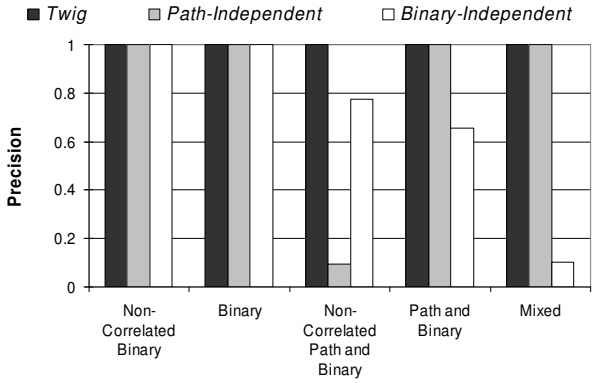

Figure 9: Precision for Datasets with various degrees of Correlation

(Figure 8) are good overall. The queries that suffer the most from the simplified *path* scoring compared to the complex *twig* scoring are those queries that have twig patterns that have branching nodes below the root node, as the correlation in these patterns is lost by the *path-independent* scoring. Note that the chain queries $q2$, $q5$ and $q7$ have low precision for one dataset: this is partly due to the fact that most of the answers to these queries are relaxed answers, and exhibit a twig pattern (due to subtree promotion), and to the presence of multiple ties in the answer set. Since our precision measure penalizes scoring approaches that produce many ties to the top-k answers, some queries exhibit low values of precision for *path-independent*. In these cases, many answers tend to be assigned high scores, resulting in low precision values, although the exact answers are part of the high scoring answers. Note that this behavior is data- and query-dependent, which is the reason why $q2$ has low precision for the medium dataset, while $q5$ and $q7$ have low precision for the large dataset. In addition, we compared *path-independent* and *twig* preprocessing times. The results are consistent with those in Figure 6, and show that *path-independent* allows for faster DAG preprocessing times than *twig* when the query does not consist of a single chain. When queries have multiple paths (or binary) predicates, the savings in preprocessing time can be significant: up to 83% for the binary query $q4$, and 72% for the twig query $q6$.

5.3.4 Effect of Correlation

We now look at the effect of data correlation on the quality of top-k answers. Figure 9 shows the precision for our scoring methods for $q3$ on datasets exhibiting different answer types; for example, the *binary* dataset only produces answers that consist of binary predicates, while the *mixed* dataset produces answers that exhibit all three predicate patterns: binary, path and twig. As expected, as soon as some of the answers have complex predicates (twig or binary), the precision of the *binary-independent* scoring method drops. Interestingly, *path-independent* precision stays equal to one for all datasets, with the notable exception of the *binary non-correlated path* dataset. Note that

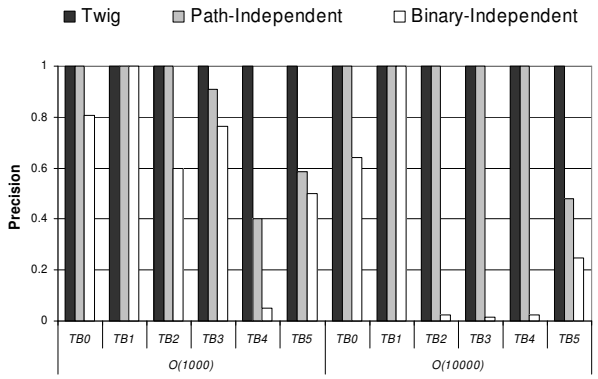

Figure 10: Precision for the Treebank Datasets

for that particular dataset, *path-independent* top-k answers have an accuracy of 64% (64% of the returned answers are part of the k top answers as returned by *twig*), however, the precision is much lower, as *path-independent* returns a high number of ties.

Path-independent has a perfect precision for datasets that produce correlated paths and twig answers. While *path-independent* does not take this correlation into account, the score ordering of the answers is not impacted, as the underlying predicate distribution is uniform. When individual path predicates have very different *idf* values, *path-independent* answers may be of low quality because the score ordering of answers may be different from that of *twig*. In effect, this means that sibling DAG nodes (DAG nodes that do not have an ancestor/descendant relationship, and therefore have no ordering constraint on their scores) may have their score ordering reversed between the *twig* and *path-independent* DAGs. We believe that this situation does not happen very often in practice.

5.3.5 Real Document Collections

Figure 10 shows the precision values for our 6 queries over the Treebank dataset. We considered two fragments of the dataset, with different sizes. Results are consistent with what we observed for synthetic data, with *binary-independent* offering low precision, and *path-independent* offering high precision, often perfect. For our real data experiments, *path-independent* lowest precision was 0.4, but in two thirds of the query tested *path-independent* exhibited perfect precision.

As future work, we plan to extend the INEX datasets and queries in order to validate our scoring methods.

6 Conclusion

We presented a family of scoring methods, inspired by the *tf*idf* approach, that account both for the structure and the content in XML documents. Our methods score relaxed answers to XML queries in a way that guarantees that the closer an answer is to the exact query, the higher is its score. We also proposed efficient implementation structures to speed up XML top-k query evaluation in this setting. We are planning to investigate streaming scenarios, where new data is constantly added to the dataset. By keeping the DAG structures for queries that users are interested in and updating the score information in a dynamic fashion we believe that we can provide an efficient and high quality top-k query answering approach for a throughput-oriented streaming framework .

References

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

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

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

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

[5] C. Delobel, 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.

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

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

[8] Y. Kanza and Y. Sagiv. Flexible Queries over Semistructured Data. PODS 2001.

[9] R. Kaushik, R. Krishnamurthy, J. Naughton and R. Ramakrishnan. On the Integration of Structure Indices and Inverted Lists. SIGMOD 2004.

[10] A. Marian, S. Amer-Yahia, N. Koudas, D. Srivastava. Adaptive Processing of Top-k Queries in XML. ICDE 2005.

[11] N. Polyzotis, M. Garofalakis, Y. Ioannidis. Approximate XML Query Answers. SIGMOD 2004.

[12] S. Robertson. The Probability Ranking Principle in IR. Journal of Documentation 33, 1977.

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

[14] D. Shin, H. Jang, H. Jin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. Proc. 3rd Int. Conf. on Dig. Lib., 1998.

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

[16] Streaming News in XML. *http://www.internetnews.com/icom_includes/feeds/inews/xml_front-10.xml, http://news.bbc.co.uk/rss/newsonline_world_edition/technology/rss091.xml*

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

[18] F. Weigel, H. Meuss, K. U. Schulz, F. Bry. Content and Structure in Indexing and Ranking XML. WebDB 2004.

[19] J. E. Wolff, H. Flörke, A. B. Cremers. Searching and Browsing Collections of Structural Information. Proc. IEEE Forum on Research and Technology Advances in Dig. Lib., 2000.