Tree Pattern Relaxation

Sihem Amer-Yahia[1], SungRan Cho[2], and Divesh Srivastava[1]

[1] AT&T Labs–Research,
Florham Park, NJ 07932, USA
{sihem,divesh}@research.att.com
[2] Stevens Institute of Technology,
Hoboken, NJ 07030, USA
scho@attila.stevens-tech.edu

Abstract. Tree patterns are fundamental to querying tree-structured data like XML. Because of the heterogeneity of XML data, it is often more appropriate to permit approximate query matching and return ranked answers, in the spirit of Information Retrieval, than to return only exact answers. In this paper, we study the problem of approximate XML query matching, based on tree pattern relaxations, and devise efficient algorithms to evaluate relaxed tree patterns. We consider weighted tree patterns, where exact and relaxed weights, associated with nodes and edges of the tree pattern, are used to compute the scores of query answers. We are interested in the problem of finding answers whose scores are at least as large as a given threshold. We design data pruning algorithms where intermediate query results are filtered dynamically during the evaluation process. We develop an optimization that exploits scores of intermediate results to improve query evaluation efficiency. Finally, we show experimentally that our techniques outperform rewriting-based and post-pruning strategies.

1 Introduction

With the advent of XML, querying tree-structured data has been a subject of interest lately in the database research community, and tree patterns are fundamental to XML query languages (e.g., [2,6,11]). Due to the heterogeneous nature of XML data, exact matching of queries is often inadequate. We believe that approximate matching of tree pattern queries and returning a ranked list of results, in the same spirit as Information Retrieval (IR) approaches, is more appropriate. A concrete example is that of querying a bibliographic database, such as DBLP [4]. Users might ask for books that have as subelements an `isbn`, a `url`, a `cdrom` and an electronic edition `ee`. Some of these are optional subelements (as specified in the DBLP schema) and very few books may have values specified for all these subelements. Thus, returning books that have values for some of these elements (say `isbn`, `url` and `ee`), as approximate answers, would be of use. Quite naturally, users would like to see such approximate answers ranked by their similarity to the user query.

C.S. Jensen et al. (Eds.): EDBT 2002, LNCS 2287, pp. 496–513, 2002.

Our techniques for approximate XML query matching are based on *tree pattern relaxations*. For example, node types in the query tree pattern can be relaxed using a type hierarchy (e.g., look for any `document` instead of just `books`). Similarly, a parent-child edge in the query tree pattern can be relaxed into an ancestor-descendant one (e.g., look for a `book` that has a descendant `isbn` subelement instead of a child `isbn` subelement). Exact matches to such relaxations of the original query are the desired approximate answers.

One possibility for ranking such approximate answers is based on the number of tree pattern relaxations applied in the corresponding relaxed query. To permit additional flexibility in the ranking (e.g., a `book` with a descendant `isbn` should be ranked higher than a `document` with a child `isbn`, even though each of these answers is based on a single relaxation), we borrow an idea from IR and consider *weighted tree patterns*. By associating exact and relaxed weights with query tree pattern nodes and edges, we allow for a finer degree of control in the scores associated with approximate answers to the query.

A query tree pattern may have a very large number of approximate answers, and returning all approximate answers is clearly not desirable. In this paper, we are interested in the problem of finding answers whose scores are at least as large as a given threshold, and we focus on the design of efficient algorithms for this problem. Our techniques are also applicable for the related problem of finding the top-k answers, i.e., the answers with the k largest scores; we do not discuss this problem further in the paper because of space limitations.

Given a weighted query tree pattern, the key problem is how to evaluate all relaxed versions of the query efficiently and guarantee that only relevant answers (i.e., those whose scores are as large as a given threshold) are returned. One possible way is to rewrite the weighted tree pattern into all its relaxed versions and apply multi-query evaluation techniques exploiting common subexpressions. However, given the exponential number of possible relaxed queries, rewriting-based approaches quickly become impractical. We develop instead (in Section 4) an algebraic representation where all our tree pattern relaxations can be encoded in a *single* evaluation plan that uses binary structural joins [1,19]. A post-pruning evaluation strategy, where all answers are computed first, and only then is pruning done, is clearly sub-optimal. Hence, we develop algorithms that eliminate irrelevant answers "as soon as possible" during query evaluation. More specifically, our technical contributions are as follows:

- We design an efficient data pruning algorithm `Thres` that takes a weighted query tree pattern and a threshold and computes all approximate answers whose scores are at least as large as the threshold (Section 5).
- We propose an adaptive optimization to `Thres`, called `OptiThres`, that uses scores of intermediate results to dynamically "undo" relaxations encoded in the evaluation plan, to ensure better evaluation efficiency, without compromising the set of answers returned (Section 6).
- Finally, we experimentally evaluate the performance of our algorithms, using query evaluation time and intermediate result sizes as metrics. Our results

validate the superiority of our algorithms, and the utility of our optimizations, over post-pruning and rewriting-based approaches (Section 7).

In the sequel, we first present related work in Section 2, and then present preliminary material in Section 3.

2 Related Work

Our work is related to the work done on keyword-based search in Information Retrieval (IR) systems (e.g., see [15]). There has been significant research in IR on indexing and evaluation heuristics that improve the query response time while maintaining a constant level of relevance to the initial query (e.g., see [7,13,18]). However, our evaluation and optimization techniques differ significantly from this IR work, because of our emphasis on tree-structured XML documents.

We classify more closely related work into the following three categories.

Language Proposals for Approximate Matching: There exist many language proposals for approximate XML query matching (e.g., see [3,8,9,12,16,17]). These proposals can be classified into content-based approaches and approaches based on hierarchical structure. In [16], the author proposes a pattern matching language called approXQL, an extension to XQL [14]. In [8], the authors describe XIRQL, an extension to XQL [14] that integrates IR features. XIRQL's features are weighting and ranking, relevance-oriented search, and datatypes with vague predicates. In [17], the authors develop XXL, a language inspired by XML-QL [6] that extends it for ranked retrieval. This extension consists of *similarity conditions* expressed using a binary operator that expresses the similarity between an XML data value and an element variable given by a query (or a constant). These works can be seen as complementary to ours, since we do not propose any query language extension in this paper.

Specification and Semantics: A query can be relaxed in several ways. In [5], the authors describe querying XML documents in a mediated environment. Their specifications are similar to our tree patterns. The authors are interested in relaxing queries whose result is empty, and they propose three kinds of relaxations: unfolding a node (replicating a node by creating a separate path to one of its children), deleting a node and propagating a condition at a node to its parent node. However, they do not discuss efficient evaluation techniques for their relaxed queries. Another interesting study is the one presented in [16] where the author considers three relaxations of an XQL query: deleting nodes, inserting intermediate nodes and renaming nodes. These relaxations have their roots in the work done in the combinatorial pattern matching community on tree edit distance (e.g., see [20]). A key difference with our work is that these works do not consider query weighting, which is of considerable practical importance.

Recently, Kanza and Sagiv [10] proposed two different semantics, flexible and semiflexible, for evaluating graph queries against a simplified version of the Object Exchange Model (OEM). Intuitively, under these semantics, query paths

are mapped to database paths, so long as the database path includes all the labels of the query path; the inclusion need not be contiguous or in the same order; this is quite different from our notion of tree pattern relaxation. They identify cases where query evaluation is polynomial in the size of the query, the database and the result (i.e., combined complexity). However, they do not consider scoring and ranking of query answers.

Approximate Query Matching: There exist two kinds of algorithms for approximate matching in the literature: *post-pruning* and *rewriting-based* algorithms. The complexity of post-pruning strategies depends on the size of query answers and a lot of effort can be spent in evaluating the total set of query answers even if only a small portion of it is relevant. Rewriting-based approaches can generate a large number of rewritten queries. For example, in [16], the rewritten query can be quadratic in the size of the original query. In our work, we experimentally show that our approach outperforms post-pruning and rewriting-based ones.

3 Overview

3.1 Background: Data Model and Query Tree Patterns

We consider a data model where information is represented as a forest of node labeled trees. Each non-leaf node in the tree has a type as its label, where types are organized in a simple inheritance hierarchy. Each leaf node has a string value as its label. A simple database instance is given in Figure 1.

Fundamental to all existing query languages for XML (e.g., [2,6,11]) are *tree patterns*, whose nodes are labeled by types or string values, and whose edges correspond to parent-child or ancestor-descendant relationships. These tree patterns are used to match relevant portions of the database. While tree patterns do not capture some aspects of XML query languages, such as ordering and restructuring, they form a key component of these query languages. Figure 1 shows an example query tree pattern (ignore the numeric labels on the nodes and edges for now). A single edge represents a parent-child relationship, and a double edge represents an ancestor-descendant relationship.

3.2 Relaxed Queries and Approximate Answers

The heterogeneity of XML data makes query formulation tedious, and exact matching of query tree patterns often inadequate. The premise of this paper is that *approximate matching of query tree patterns and returning a ranked list of answers, in the same spirit as keyword-based search in Information Retrieval (IR)* is often more appropriate.

Our techniques for approximate XML query matching are based on *tree pattern relaxations*. Intuitively, tree pattern relaxations are of two types: content relaxation and structure relaxation. We consider four specific relaxations, of which the first two are content relaxations, and the last two are structure relaxations.

Node Generalization: This permits the type of a query node to be generalized to a super-type. For example, in the query tree pattern of Figure 1, `Book` can be generalized to `Document`, allowing for arbitrary documents (that match the other query conditions) to be returned instead of just books.

Leaf Node Deletion: This permits a query leaf node (and the edge connecting it to its parent node in the query) to be deleted. For example, in the query tree pattern of Figure 1, the `Collection` node can be deleted, allowing for books that have an editor (with a name and address) to be returned, whether or not they belong to a collection.

Edge Generalization: This permits a parent-child edge in the query to be generalized to an ancestor-descendant edge. For example, in the query tree pattern of Figure 1, the edge (`Book`, `Editor`) can be generalized, allowing for books that have a descendant editor (but not a child editor) to be returned.

Subtree Promotion: This permits a query subtree to be promoted so that the subtree is directly connected to its former grandparent by an ancestor-descendant edge. For example, in the query tree pattern of Figure 1, the leaf node `Address` can be promoted, allowing for books that have a descendant address to be returned, even if the address does not happen to be a descendant of the editor child of the book.

Having identified the individual tree pattern relaxations we consider, we are now in a position to define relaxed queries and approximate answers.

Definition 1 [Relaxed Query, Approximate Answer]. *Given a query tree pattern Q, a* relaxed query *Q' is a non-empty tree pattern obtained from Q by applying a sequence of zero or more of the four relaxations: node generalization, leaf node deletion, edge generalization and subtree promotion.*

We refer to a node (resp., edge) in a relaxed query that has been affected by a tree pattern relaxation as a relaxed node *(resp.,* relaxed edge*). The nodes and edges that are not affected by a tree pattern relaxation are referred to as* exact nodes *and* exact edges.

An approximate answer *to Q is defined as an exact match to some relaxed query obtained from Q.* □

Note that, by definition, the original query tree pattern is also a relaxed query, and hence exact matches to the original query tree pattern are included in the set of approximate answers to a query.

Note that the tree relaxations we consider have several interesting properties. First, the number of nodes in a relaxed query is no more than in the original query. Second, an answer to the original query continues to be an answer to a relaxed query. Finally, each individual tree pattern relaxation is local, involving either a single node/edge change (in the cases of node generalization and edge generalization), or two changes (in the cases of leaf node deletion and subtree promotion). These properties will serve as the bases for efficient algorithms for the computation of approximate answers.

3.3 Answer Ranking, Weighted Tree Patterns and Answer Scores

Returning approximate answers in ranked order, based on the extent of approximation, is important, as is evident from IR research and web search engines. One possibility for ranking such approximate answers is based on the *number* of tree pattern relaxations present in the corresponding relaxed query, i.e., all answers corresponding to relaxed queries with the same number of relaxations have the same rank. While such a coarse ranking may suffice for some applications, additional flexibility is typically desirable. For this purpose, we consider weighted tree patterns, defined as follows.

Definition 2 [Weighted Tree Pattern]. *A* weighted tree pattern *is a tree pattern where each node and edge is assigned two non-negative integer weights: an exact weight ew, and a relaxed weight rw, such that $ew \geq rw$.* □

Figure 1 shows an example of a weighted query tree pattern. A detailed discussion of the origin of query weights is outside the scope of this paper. It may be specified by the user, determined by the system (e.g., in a fashion analogous to inverse document frequency, used in IR), or a combination of both. What is important to keep in mind is that once these weights are chosen, our techniques can be used for efficient computation of approximate answers.

Relaxation of a weighted query tree pattern results in a weighted tree pattern as well. The weights on nodes and edges in a relaxed query Q' are used to determine scores of the corresponding matches, by adding up the contributions of the individual nodes and edges in Q', as follows:

- The contribution of an exact node or edge, ne, in Q' to the score of an exact match A' to Q' is its exact weight $ew(ne)$.
- The contribution of a relaxed node or edge, ne, in Q' to the score of an exact match A' to Q' is required to be no less than its relaxed weight $rw(ne)$, and no more than its exact weight $ew(ne)$.

A simple approach, which we use in our examples and our experiments, is to make the relaxed weight $rw(ne)$ be the contribution of the relaxed node or edge ne. More sophisticated alternatives are possible as well. We do not discuss these further for reasons of space.

As an example, the score of exact matches of the weighted query tree pattern in Figure 1 is equal to the sum of the exact weights of its nodes and edges, i.e., 45. If `Book` is generalized to `Document`, the score of an approximate answer that is a document (but not a book) is the sum of the relaxed weight of `Book` and the exact weights of the other nodes and edges in the weighted query, i.e., 39.

In general, an approximate answer can match different relaxed queries, and, depending on how one defines the contributions due to relaxed nodes and edges, may end up with different scores. To deal with such a situation, we define the score of an approximate answer as follows.

Definition 3 [Score of an Approximate Answer]. *The* score *of an approximate answer is the maximum among all scores computed for it.* □

We are now finally ready to define the problem that we address in this paper.

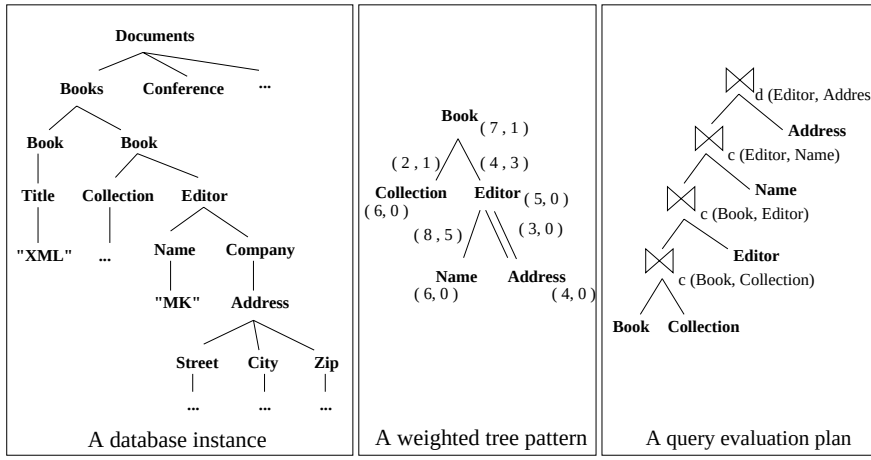


Fig. 1. Example Database Instance, Weighted Tree Pattern, Query Evaluation Plan

3.4 Problem Definition

A query tree pattern may, in general, have a very large number of approximate answers, and returning all approximate answers to the user is clearly not desirable. In this paper, we focus on an approach to limiting the number of approximate answers returned based on a threshold.

Definition 4 [Threshold Problem]. *Given a weighted query tree pattern Q and a threshold t, the* threshold problem *is that of determining all approximate answers of Q whose scores are $\geq t$.* □

4 Encoding Relaxations in a Query Evaluation Plan

4.1 Query Evaluation Plan

Several query evaluation strategies have been proposed for XML (e.g., [11,19]). They typically rely on a combination of index retrieval and join algorithms using specific structural predicates. For the case of tree patterns, the evaluation plans make use of two binary structural join predicates: $c(n_1, n_2)$ to check for the parent-child relationship, and $d(n_1, n_2)$ to check for the ancestor-descendant one.

The query evaluation techniques we have developed (and will present in subsequent sections), for efficiently computing approximate answers, rely on the use of such join plans to evaluate tree patterns.[1] Figure 1 shows a translation of the (unweighted) query tree pattern of Figure 1 into a left-deep, join evaluation plan with the appropriate structural predicates. According to this evaluation plan, an

[1] However, our techniques are not limited to using a particular join algorithm, even though we use the stack-based join algorithms of [1] in our implementation.

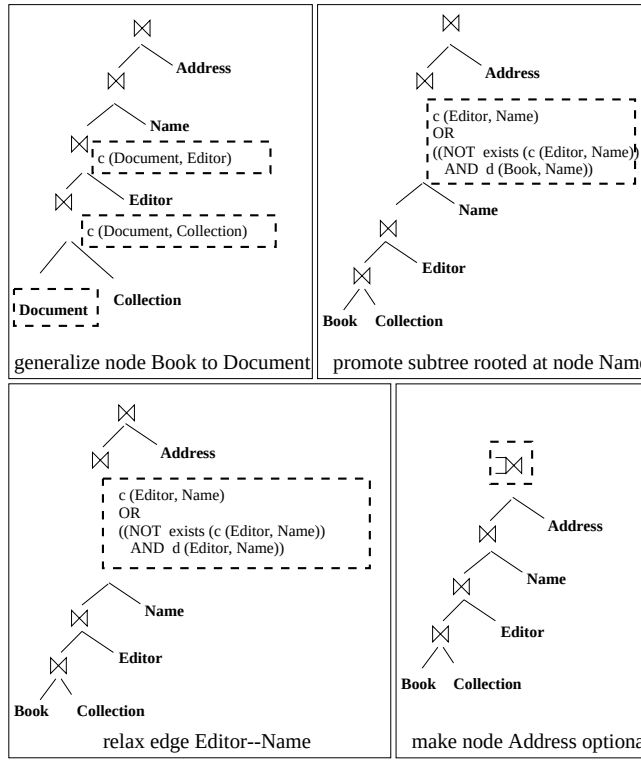


Fig. 2. Encoding Individual Tree Pattern Relaxations

answer to a query is an n-tuple containing a node match for every leaf node in the evaluation plan (i.e., for every node in the query tree pattern).

4.2 Encoding Tree Pattern Relaxations

We show how tree pattern relaxations can be encoded in the evaluation plan. Figure 2 presents some example relaxations of the (unweighted) query tree pattern of Figure 1, and specifies how the query evaluation plan of Figure 1 needs to be modified to encode these relaxations. The modifications to the initial evaluation plan are highlighted with bold dashed lines. Predicates irrelevant to our discussion are omitted.

Node Generalization: In order to encode a node generalization in an evaluation plan, each predicate involving the node type is replaced by a predicate on its super-type. For example, Figure 2 depicts how `Book` can be generalized to `Document` in the evaluation plan.

Edge Generalization: In order to capture the generalization of a parent-child edge to an ancestor-descendant edge in an evaluation plan, we transform the

join predicate $c(\tau_1, \tau_2)$ into the predicate:

$$c(\tau_1, \tau_2) \text{ OR } ((\not\exists\, c(\tau_1, \tau_2)) \text{ AND } d(\tau_1, \tau_2))$$

This new join predicate can be checked by first determining if a parent-child relationship exists between the two nodes, and then, if this relationship doesn't exist, determining if an ancestor-descendant relationship exists between them. For example, Figure 2 depicts how the parent-child edge (`Editor`, `Name`) can be generalized to an ancestor-descendant edge in the evaluation plan.

In subsequent figures, this predicate is simplified to $(c(\tau_1, \tau_2)$ OR $d(\tau_1, \tau_2))$, where the OR has an ordered interpretation (check $c(\tau_1, \tau_2)$ first, $d(\tau_1, \tau_2)$ next).

Leaf Node Deletion: To allow for the possibility that a given query leaf node may or may not be matched, the join that relates the leaf node to its parent node in the query evaluation plan becomes an outer join. More specifically, it becomes a left outer join for left-deep evaluation plans. For example, Figure 2 illustrates how the evaluation plan is affected by allowing the `Address` node to be deleted. The left outer join guarantees that even books whose editor does not have an address will be returned as an approximate answer.

Subtree Promotion: This relaxation causes a query subtree to be promoted to become a descendant of its current grandparent. In the query evaluation plan, the join predicate between the parent of the subtree and the root of the subtree, say $jp(\tau_1, \tau_2)$ needs to be modified to:

$$jp(\tau_1, \tau_2) \text{ OR } ((\not\exists\, jp(\tau_1, \tau_2)) \text{ AND } d(\tau_3, \tau_2))$$

where τ_3 is the type of the grandparent. For example, Figure 2 illustrates how the evaluation plan is affected by promoting the subtree rooted at `Name`.

Again, in subsequent figures, this new join predicate is simplified to $(c(\tau_1, \tau_2)$ OR $d(\tau_3, \tau_2))$, where the OR has an ordered interpretation.

Combining Relaxations: Figure 3(a) shows the evaluation plan obtained by encoding all possible tree pattern relaxations of the query tree pattern of Figure 1. Each node is generalized if a type hierarchy exists (in our example query, only `Book` becomes `Document`). All parent-child edges are generalized to ancestor-descendant edges. All nodes, except the tree pattern root, are made optional. Finally, all subtrees are promoted. Note that even non-leaf nodes such as `Editor` can be deleted once its subtrees are promoted, and it becomes a leaf node.

5 An Efficient Solution to the Threshold Problem

The goal of the threshold approach is to take a weighted query tree pattern and a threshold, and generate a ranked list of approximate answers whose scores are at least as large as the threshold, along with their scores. A simple approach to achieve this goal is to (i) translate the query tree pattern into a join evaluation

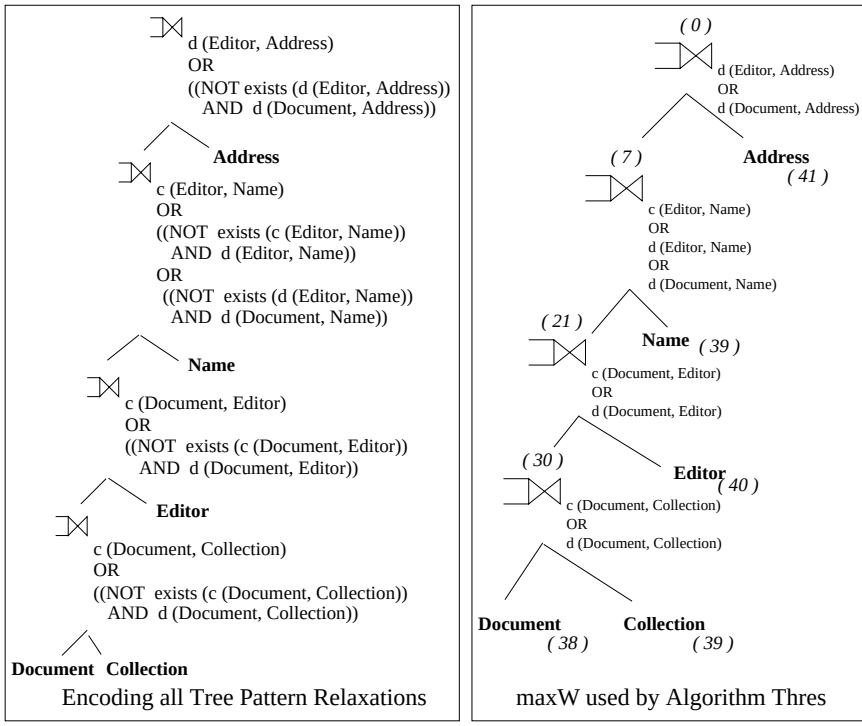


Fig. 3. Encoding All Relaxations, Weights used by `Thres`

plan, (ii) encode all possible tree pattern relaxations in the plan (as described in Section 4), (iii) evaluate the modified query evaluation plan to compute answers to all relaxed queries (along with their scores), and (iv) finally, return answers whose scores are at least as large as the threshold. We show that this *post-pruning* approach is suboptimal since it is not necessary to first compute all possible approximate answers, and only then prune irrelevant ones.

In order to compute approximate answers more efficiently, we need to detect, as soon as possible during query evaluation, which intermediate answers are guaranteed to not meet the threshold. For this purpose, we took inspiration from evaluation algorithms in IR for keyword-based searches, and designed Algorithm `Thres`. `Thres` operates on a join evaluation plan. Before describing this algorithm, we discuss an example to illustrate how approximate answer scores are computed at each step of the join evaluation plan.

5.1 Computing Answer Scores: An Example

The following algebraic expression (a part of the join plan of Figure 3) illustrates the types of results computed during the evaluation of the join plan:

$$\texttt{Document} \ \overline{\bowtie}_{c(\texttt{Document},\texttt{Collection}) \ \texttt{OR} \ d(\texttt{Document},\texttt{Collection})} \ \texttt{Collection}$$

Suppose that the node `Document` is a generalization of an initial node `Book`. Evaluating `Document` (say, using an index on the node type) results in two kinds of answers: (i) answers whose type is the exact node type `Book`, and (ii) answers whose type is the relaxed node type `Document`, but not `Book`. An answer in the first category is assigned the exact weight of this node as its score, i.e., 7. An answer in the second category is assigned the (typically smaller) relaxed weight as its score, i.e., 1.

Let *doc* denote answers of type `Document` (with score s_1) and *col* denote answers of type `Collection` (with score s_2). The result of the above algebraic expression includes three types of answers:

- (doc, col) pairs that satisfy the structural predicate $c(doc, col)$.
- (doc, col) pairs that do not satisfy $c(doc, col)$, but satisfy the structural predicate $d(doc, col)$.
- *doc*'s that do not join with any *col* via $c(doc, col)$ or $d(doc, col)$.

The score of a (doc, col) pair is computed as $s_1 + s_2 + s(doc, col)$, where $s(doc, col)$ is the contribution due to the edge between `Document` and `Collection` in the query (see Section 3.3 for more details). The score of a *doc* that does not join with any *col* is s_1.

5.2 Algorithm `Thres`

The basis of Algorithm `Thres`, which prunes intermediate answers that cannot possibly meet the specified threshold, is to associate with each node in the join evaluation plan, its *maximal weight*, `maxW`, defined as follows.

Definition 5 [Maximal Weight]. *The* maximal weight*,* `maxW`*, of a node in the evaluation plan is defined as the largest value by which the score of an intermediate answer computed for that node can grow.* □

Consider, for example, the evaluation plan in Figure 3. The `maxW` of the `Document` node is 38. This number is obtained by computing the sum of the exact weights of all nodes and edges of the query tree pattern, excluding the `Document` node itself. Similarly, `maxW` of the join node with `Editor` as its right child is 21. This is obtained by computing the sum of the exact weights of all nodes and edges of the query tree pattern, excluding those that have been evaluated as part of the join plan of the subtree rooted at that join node. By definition, `maxW` of the last join node, the root of the evaluation plan, is 0.

Algorithm `Thres` is summarized in Figure 4. It needs `maxW` to have been computed at each node of the evaluation plan. The query evaluation plan is executed in a bottom-up fashion. At each node, intermediate results, along with their scores, are computed. If the sum of the score of an intermediate result and `maxW` at the node does not meet the threshold, this intermediate result is eliminated. Note that Figure 4 shows a nested loop join algorithm for simplicity of exposition. The algorithms we use for inner joins and left outer joins are based on the structural join algorithms of [1].

```
Algorithm Thres(Node n)
   if (n is leaf) {
      list = evaluateLeaf(n);
      for (r in list)
         if (r->score + n->maxW ≥ threshold)  append r to results;
      return results; }
   list1 = Thres(n->left);
   list2 = Thres(n->right);
   for (r1 in list1) {
      for (r2 in list2) {
         if (checkPredicate(r1,r2,n->predicate))
            s = computeScore(r1,r2,n->predicate);
         if (s + n->maxW ≥ threshold)
            append (r1,r2) to results with score s; }
      if (∄ r2 that joins with r1)
         if (r1->score + n->maxW ≥ threshold)
            append (r1,-) to results with score r1->score;
   }
   return results;
```

Fig. 4. Algorithm `Thres`

6 An Adaptive Optimization Strategy

6.1 Algorithm `OptiThres`

The key idea behind `OptiThres`, an optimized version of `Thres`, is that we can predict, during evaluation of the join plan, if a subsequent relaxation produces additional matches that will *not* meet the threshold. In this case, we can "undo" this relaxation in the evaluation plan. Undoing this relaxation (e.g., converting a left outer join back to an inner join, or reverting to the original node type) improves efficiency of evaluation since fewer conditions need to be tested and fewer intermediate results are computed during the evaluation.

While Algorithm `Thres` relies on `maxW` at each node in the evaluation plan to do early pruning, Algorithm `OptiThres` additionally uses three weights at each join node of the query evaluation plan:

- The first weight, `relaxNode`, is defined as the largest value by which the score of an intermediate result *computed for the left child of the join node* can grow if it joins with a relaxed match to the right child of the join node. This is used to decide if the node generalization (if any) of the right child of the join node should be unrelaxed.
- The second weight, `relaxJoin`, is defined as the largest value by which the score of an intermediate result *computed for the left child of the join node* can grow if it cannot join with any match to the right child of the join node.

```
Algorithm OptiThres(Node n)
   if (n is leaf) {
      // evaluate, prune, and return results as in Algorithm Thres
   }
   list1 = OptiThres(n->left);
   /* maxLeft is set to the maximal score of results in list1 */
   if (maxLeft + relaxNode < threshold)  unrelax(n->right);
   list2 = OptiThres(n->right);
   /* maxRight is set to the maximal score of results in list2 */
   if (maxLeft + relaxJoin < threshold)  unrelax(n->join);
   if (maxLeft + maxRight + relaxPred < threshold)
      unrelax(n->join->predicate);
   // now, evaluate, prune and return join (and possibly outer join)
   // results as in Algorithm Thres
```

Fig. 5. Algorithm `OptiThres`

This is used to decide if the join node should remain a left outer join, or should go back to being an inner join.

– The third weight, `relaxPred`, is defined as the largest value by which the sum of the scores of a pair of intermediate results for the left and right children of the join node can grow if they are joined using a relaxed structural predicate. This is used to decide if the edge generalization and subtree promotion should be unrelaxed.

Algorithm `OptiThres` is given in Figure 5. Only the parts that are modifications to Algorithm `Thres` are specified, and we indicate where the code fragments from Algorithm `Thres` need to be inserted. It is easy to see that `OptiThres` has very few overheads over Algorithm `Thres`, since `OptiThres` makes use of the maximal score of answers at each step of the evaluation process (which can be maintained in limited space while the intermediate answers are being computed), and some precomputed numbers at each node in the evaluation plan.

Finally, note that Algorithm `OptiThres` makes only *local* decisions about undoing relaxations (not generalizing the right child of the join, turning the outer join to an inner join, or turning the join predicate from descendant to child). A natural question is whether a more global approach could do better. It is not too difficult to see that applying `OptiThres` locally at each node is at least as good as applying it globally since a global optimization would have to rely on more conservative estimates of possible scores of intermediate results.

6.2 An Illustrative Example

We illustrate Algorithm `OptiThres` using an example. Consider the weighted query tree pattern in Figure 6. This query looks for all `Proceedings` that have as children subelements a `Publisher` and a `Month`. Exact and relaxed weights

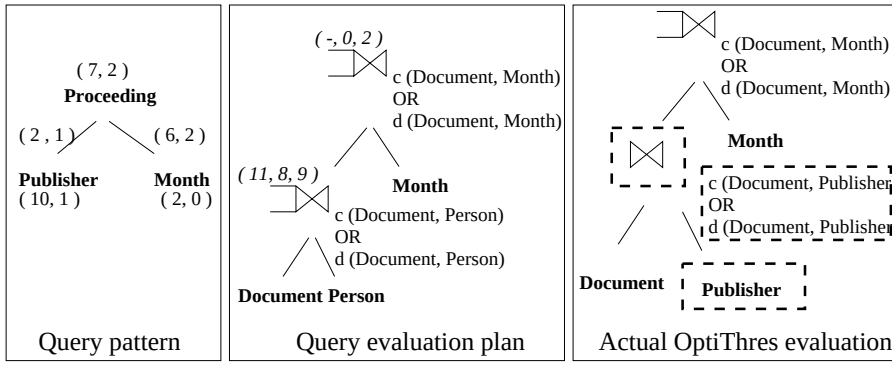


Fig. 6. A Simple `OptiThres` Example

are associated with each node and edge in the query tree pattern. `Proceeding` is relaxed to `Document`, `Publisher` is relaxed to `Person`, the parent-child edges are relaxed to ancestor-descendant ones, and nodes `Person` and `Month` are made optional. The threshold is set to 14.

First, weights are computed at each evaluation plan node statically. Recall that in our examples, we have chosen to use the relaxed weight as the contribution due to matching a relaxed node or edge. For example, at the first join node in the evaluation plan, `relaxNode` = 11 (ew(`Month`) + ew(`(Proceeding, Month)`) + ew(`(Proceeding, Publisher)`) + rw(`Publisher`)), `relaxJoin` = 8 (ew(`Month`) + ew(`(Proceeding, Month)`)), and `relaxPred` = 9 (ew(`Month`) + ew(`(Proceeding, Month)`) + rw(`(Proceeding, Publisher)`)).

Next, Algorithm `OptiThres` evaluates the annotated query evaluation plan in Figure 6. `Document` is evaluated first. Assume that the maximal score in the list of answers we get is 2, i.e., there are no `Proceeding`'s in the database. At the next join, `relaxNode` = 11, `relaxJoin` = 8, and `relaxPred` = 9. The sum `relaxNode` +2 = 13, which is smaller than the threshold. In this case, `OptiThres` decides to unrelax `Person` to `Publisher`, and the plan is modified suitably. Next, `Publisher` is evaluated, and let the maximal score in the result list be 10 (i.e., exact matches were obtained). The sum `relaxJoin` +2 = 10, which is also smaller than the threshold, and `OptiThres` decides to unrelax the left outer join to an inner join, since we cannot "afford to lose `Publisher`". The algorithm then checks whether to retain the descendant structural predicate. Since the sum `relaxPred` +2 + 10 = 21, which is larger than the threshold, `OptiThres` decides to retain the relaxed structural join predicate d(`Document, Publisher`).

During the evaluation of the first join, join results are pruned using `maxW`, as in Algorithm `Thres`. Assume that the maximal score of answers in the first join result is 14 (10+2+2). `OptiThres` then uses the weights at the second join node to determine whether any other relaxations need to be undone. Note that `Month` node has not been generalized, and this is reflected in the fact that `relaxNode` at the second join is not specified. Next, `Month` is evaluated, and matches have

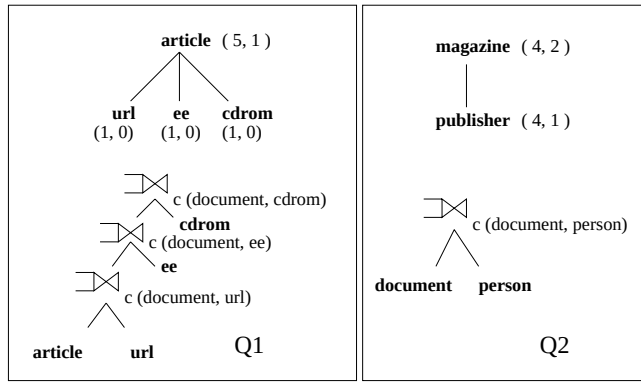


Fig. 7. Queries Used in the Experiments

a score of 2. The sum `relaxJoin` +14 = 14, which meets the threshold. So the outer join is not unrelaxed. Similarly, `relaxPred` +14 + 2 = 18, which meets the threshold. So the join predicate is not unrelaxed. Finally, the second join is evaluated, and join results are pruned using `maxW`, as in Algorithm `Thres`.

The algebraic expression that we have effectively computed is given in Figure 6, where the dynamically modified portions of the evaluation plan are highlighted.

7 Experiments

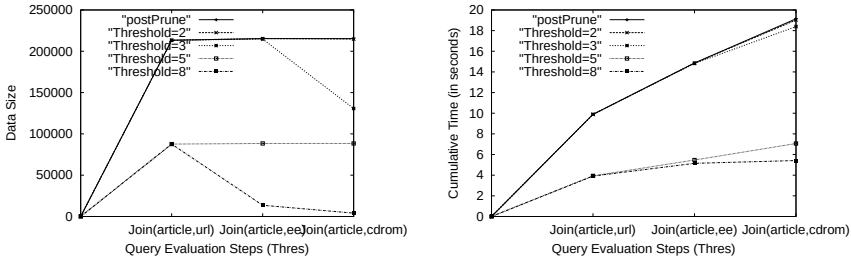
7.1 Experimental Setup

We use the `DBLP` XML dataset which is approximately 85MBytes and contains 2.1M elements. Some information about relevant elements is given in the table below. The `DTD` of this dataset as well as the data itself can be found at `http://dblp.uni-trier.de/db`.

Label	No. of elements	Label	No. of elements
article	87,675	url	212,792
cdrom	13,052	ee	55,831
document	213,362	magazine	0
publisher	1,199	person	448,788

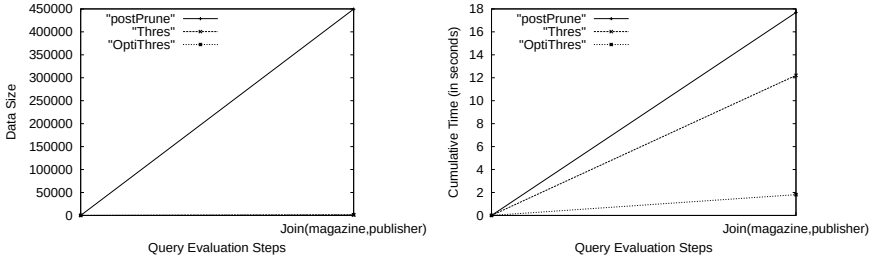
In our type hierarchy, `document` is a super-type of `book`, `incollection`, `inproceedings`, `proceedings`, `article`, `phdthesis`, `mastersthesis`, `www` and `magazine`; and `person` is a super-type of `author`, `editor` and `publisher`. We use the queries of Figure 7. Since the DTD does not have long root-to-leaf paths, we do not consider edge generalization and subtree promotion in our experiments.

In order to prune data at each step of the query evaluation, we modified the stack-based structural join algorithm of [1] so that each input (and output) is



(a) Data Size Comparison (b) Time Comparison

Fig. 8. Comparing `Thres` and `postPrune`



(a) Data Size Comparison (b) Time Comparison

Fig. 9. Comparing `OptiThres`, `Thres` and `postPrune`

materialized in a file. We ran all our experiments on a HP-UX machine with 32MBytes of memory. In all experiments, query evaluation time is reported in seconds and result sizes in the number of answers.

7.2 Studying Algorithm `Thres`

We use **Q1** where `url`, `ee`, and `cdrom` are made optional and `article` is relaxed to `document`. We compare (i) the evaluation times for `Thres` (for multiple thresholds) and `postPrune`, and (ii) the cumulative sizes of the data processed by each algorithm. The results are given in Figure 8 where the X-axis represents each step of **Q1** evaluation. Figure 8(a) shows that the higher the threshold, the earlier is irrelevant data pruned and the smaller is the evaluation time. This is explained by the fact that with a higher threshold, the amount of data that remains in the evaluation process is reduced (as shown in Figure 8(b)). For `postPrune`, data pruning occurs only at the last step of query evaluation.

7.3 Benefit of Algorithm `OptiThres`

We compare `postPrune`, `Thres` and `OptiThres`. We use query **Q2** with a threshold = 5 because we want to illustrate how `OptiThres` decides that `publisher` should not be relaxed to `person`. `magazine` is relaxed to `document`, `publisher` to `person`, and `person` is made optional. Figure 9 shows an intermediate data size and an evaluation time comparison. `OptiThres` detects that `publisher` should not have been relaxed to `person` and should not have been made optional (the outer join is turned back to an inner join). This is because there is no `magazine` in the database, and the only instances that are selected when evaluating `document` are documents that are not magazines. Thus, their scores are `2` and would not meet the threshold if `publisher` is relaxed. The graphs of Figure 9(a) show that both `postPrune` and `Thres` scan all of `person` which results in processing more data than `OptiThres` (which scans only `publisher`). This also results in a higher evaluation time as shown in Figure 9(b). In addition, since `OptiThres` performs an inner join operation (instead of an outer join), there are evaluation time and data size savings at the last step of query evaluation.

Since `OptiThres` prunes data earlier than the other strategies, it manipulates the least amount of data, and thus its evaluation time is the smallest. Due to its ability to undo unnecessary relaxations, `OptiThres` achieves a significant improvement in query evaluation performance.

7.4 Comparison with Rewriting-Based Approaches

We run all our algorithms on query `Q1` with a threshold set to 2 (to select a large number of answers). We compare `postPrune`, `OptiThres`, `MultiQOptim` and `MultiQ`. `MultiQ` and `MultiQOptim` are two rewriting-based approaches. `MultiQ` is the case where we generate all relaxed versions of `Q1` and execute each of them separately. The total evaluation time is obtained by adding each of their evaluation times. `MultiQOptim` is the case where we share common subexpressions.

`postPrune` took 22.384 seconds, `OptiThres` took 18.550, `MultiQOptim` took 30.782 and `MultiQ` took 40.842. Our results show that the execution time of `OptiThres` is considerably faster than rewriting-based approaches. The reason is that `MultiQ` performs 10 joins, `MultiQOptim` performs 8 joins and `OptiThres` performs only 3 joins.

8 Conclusion

In this paper we have developed techniques for relaxing weighted query tree patterns, and efficiently computing approximate answers to weighted tree patterns by encoding the relaxations in join evaluation plans. Our preliminary experimental evaluation has shown the benefits of our techniques over post-pruning and rewriting-based approaches.

There are many interesting directions of future work in this area. What is the analog of the $tf * idf$ score used in Information Retrieval for keyword-based

queries? How can one combine our optimizations with traditional cost-based join ordering to identify the cheapest evaluation plan? How does one quickly estimate the number of approximate answers that meet a given threshold? Solutions to these problems will be important for the XML database systems of tomorrow.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: Efficient matching of XML query patterns. In *Proceedings of ICDE*, 2002.
2. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery.
3. Y. Chiaramella, P. Mulhem, and F. Fourel. A model for multimedia information retrieval. Technical report, FERMI ESPRIT BRA 8134, University of Glasgow. www.dcs.gla.ac.uk/fermi/tech_reports/reports/fermi96-4.ps.gz.
4. DBLP Database. http://www.informatik.uni-trier.de/ ley/db.
5. C. Delobel and M. C. Rousset. A uniform approach for querying large tree-structured data through a mediated schema. In *Proceedings of International Workshop on Foundations of Models for Information Integration (FMII)*, 2001.
6. A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of WWW*, 1999.
7. C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1), 1985.
8. N. Fuhr and K. Grossjohann. XIRQL: An extension of XQL for information retrieval. In *Proceedings of SIGIR*, 2001.
9. Y. Hayashi, J. Tomita, and G. Kikui. Searching text-rich XML documents with relevance ranking. In *Proceedings of SIGIR Workshop on XML and Information Retrieval*, 2000.
10. Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of PODS*, 2001.
11. J. McHugh, S Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. SIGMOD Record, 26(3):45-66, 1997.
12. S. Myaeng, D.-H. Jang, M.-S. Kim, and Z.-C. Zhoo. A flexible Model for retrieval of SGML documents. In *Proceedings of SIGIR*, 1998.
13. M. Persin. Document filtering for fast ranking. In *Proceedings of SIGIR*, 1994.
14. J. Robie, J. Lapp, and D. Schach. XML query language (XQL). Available from http://www.w3.org/TandS/QL/QL98/pp/xql.html.
15. G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, New York, 1983.
16. T. Schlieder. Similarity search in XML data using cost-based query Transformations. In *Proceedings of SIGMOD WebDB Workshop*, 2001.
17. A. Theobald and G. Weikum. Adding relevance to XML. In *Proceedings of SIGMOD WebDB Workshop*, 2000.
18. H. Turtle and J. Flood. Query evaluation: Strategies and optimization. *Information Processing & Management*, Nov. 1995.
19. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.
20. K. Zhang and D. Shasha. Tree pattern matching. *Pattern Matching Algorithms, Apostolico and Galil (Eds.), Oxford University Press*, 1997.