PaperNumber: 187

Tree Pattern Relaxation

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

SungRan Cho
Stevens Institute of Technology
scho@attila.stevens-tech.edu

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

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 exact matches. 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 pattern, are used to compute the scores of query answers. We are interested in two problems: (i) finding answers whose scores are at least as large as a given threshold, and (ii) finding the top-k answers. We design data pruning algorithms where intermediate query results are filtered dynamically during the evaluation process. Finally, we show experimentally that our approach outperforms rewriting-based and post-pruning strategies. We develop several optimizations that exploit scores of intermediate results to improve query evaluation efficiency.

1 Introduction

With the advent of XML, querying tree-structured data has been a subject of interest lately in the database research community (e.g. [8, 14, 22]). Due to the heterogeneous nature of XML data, exact matching of queries is often inadequate. We believe that approximate matching of tree 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 the DBLP database [12]. 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`) would be of use. In fact, users might themselves specify that they are interested in the k most relevant answers in which case answers should be returned ranked by their similarity to the exact query.

In IR [28], search techniques process keyword-based queries by (i) associating weights to query keywords (e.g. based on inverse document frequency), (ii) computing scores of documents based on the occurrence of query keywords in them and, (iii) returning answer documents ranked on their scores. IR techniques for processing keywords are certainly useful when matching tree pattern queries approximately against XML documents. However, they are not sufficient and incorporating techniques for *the approximate matching of the document tree structure* is more appropriate for XML documents.

In order to compute approximate matches to a tree pattern query, this query must be *relaxed*. Tree pattern relaxation can be achieved in several ways. For example, node types in the query can be generalized using a type hierarchy (e.g., look for any `document` instead of only `books`). Also, child relationships can be transformed into descendant ones (e.g., look for a descendant `isbn` subelement of the `book` instead of just as a child of `book`). Users can specify the maximal number of approximate answers they are expecting or a threshold that constitutes a lower bound for relevant query answers. These criteria can be used to prune irrelevant query matches. In this paper, we focus on the design of efficient techniques to perform these approximate matchings on tree pattern queries.

Given a tree pattern, the key problem is *how to evaluate all relaxed versions of the query efficiently and guarantee that only relevant answers (those that are in the top-k list or those whose score is at least as large as a given threshold) are returned.* One possible way of evaluating a relaxed query is to rewrite it into all its relaxed versions and apply multi-query evaluation techniques exploiting common subexpressions. However, given the number of possible relaxations, rewriting-based approaches quickly become impractical. Using techniques inspired by IR is a more promising approach. All possible query relaxations can be encoded in a single query evaluation plan which is evaluated and only relevant answers are selected. A post-pruning evaluation strategy could be used but it is sub-optimal because it requires to compute all answers first, then perform pruning. Hence, we developed algorithms that eliminate irrelevant answers "as soon as possible" in the query evaluation process. More specifically, our technical contributions are as follows:

- We design an efficient data pruning algorithm `Thres` that takes a weighted query and a threshold and computes a set of answers whose score is at least as large as this threshold.
- `Thres` applies all possible relaxations to a query and then eliminates answers that have been relaxed "too much" based on the threshold. We propose an *adaptive optimization* to `Thres`, called `OptiThres`, that uses scores of intermediate results to detect dynamically which relaxations should be undone, while a query is being evaluated, to ensure better efficiency.
- We design `Top-K`, an algorithm based on the same idea as `Thres` to select the top-k answers to a query. `Top-K` takes a weighted query and a number k and returns a ranked list of the k answers with the highest score.
- Finally, we run a set of experiments to compare the performance of our algorithms. The experiments show the evaluation time and data size trends of each algorithm. They validate the superiority of our algorithms when compared to post-pruning and rewriting-based ones. They also study the overhead of relaxing a query compared to finding exact matches only. Finally, they validate the various optimizations that we propose.

In the sequel, we first present the related work in Section 2. The following section (Section 3) contains the background of this study (data model and queries). It also gives the semantics of our work and defines the problems we are tackling. Section 4 describes query evaluation plans and how relaxations are encoded in a query plan. Then, in Section 5, we describe `Thres` and `OptiThres`. `Top-K` is given in Section 6. Finally, Section 7 presents a set of experimental results. The terms "query pattern", "tree query" and "tree pattern" will be used interchangeably in this paper.

2 Related Work

Our work focuses on approximate answering of tree pattern queries, which is closely related to approximate keyword matching in Information Retrieval (IR) systems [28]. There has been significant research in IR on indexing techniques and query evaluation heuristics that improve the query response time while maintaining a constant level of relevance to the initial query (e.g., see [5, 18, 26, 31, 32, 34]). In fact, our strategy of using data pruning for obtaining answers to relaxed tree patterns was inspired by the work done in IR. However, our evaluation and optimization techniques differ considerably from the IR work, because of our emphasis on tree-structured XML documents.

In order to understand our contributions, we classify the related work into the following four categories.

Language Proposals for Approximate Matching. There exist many language proposals for approximate query matching. These proposals can be classified into two main categories: *content-based* approaches and approaches based on *hierarchical structure*. In the first category, we find text search and extensions to it for querying *position* of text (using predicates such as *near*) in documents (e.g, [11, 19, 21, 24, 29, 30]). In the second category, we find [29].

In [29], the author proposes a pattern matching language called approXQL, an extension to XQL [27]. In [19], the authors describe XIRQL, an extension to XQL [27] that integrates IR features. XIRQL's features are weighting and ranking, relevance-oriented search (where only the requested content is specified and not the type of elements to be retrieved) and datatypes with vague predicates (e.g., search for measurements that were taken at about 30 feet). In [30], the authors develop XXL, a language inspired by XML-QL [14] that extends it for ranked retrieval. This extension consists of *similarity conditions* expressed using a binary operator that expresses the similarity between a value of a node of the XML data tree and a constant or an element variable given by a query. This operator can also be used for approximate matching of element and attribute names.

In this paper, we do not propose any query language extension and the works described above can be seen as complementary to ours.

Specification and Semantics. A query can be relaxed in several ways. In [13], the authors describe querying XML documents in a mediated environment. The query language is similar to our tree patterns. The authors are interested in relaxing queries whose result is empty. 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. Unfortunately, this work does not consider any weighting and does not discuss evaluation techniques for relaxed queries. Another interesting work is the one presented in [29] where the author considers three relaxations of an XQL [27] query: deleting nodes, inserting intermediate nodes and renaming nodes. By allowing only stylized sequences of deleting nodes (in a bottom-up fashion), [29] avoids the combinatorial effects of permitting arbitrary combinations of deletions.

Recently, Kanza and Sagiv [23] 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.

In IR, there are three ways of controlling the set of relaxations that are applied to a query: *threshold*, *top-k* and *boolean* (e.g., [21] and [30]) approaches. Most often, query terms are assigned weights based on some variant of the tf*idf method [28] and probability independence between elementary conditions is assumed. We do not focus on the computation of the weights of query terms and we can use existing methods for weight estimation.

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 very large number of rewritten queries. For example, in [29], 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.

Top-k Query Processing. Relevant work on top-k query processing can roughly be divided in two groups: evaluation strategies for multiattribute queries over multimedia repositories, and evaluation strategies for queries over relational databases.

To process top-k queries involving multiple attributes, Fagin proposed the FA algorithm [16], which was developed as part of IBM Almaden's Garlic project. This algorithm can evaluate top-k queries that involve several independent "multimedia subsystems", each producing scores that are combined using arbitrary monotonic aggregation functions. Fagin showed that this technique is optimal in a probabilistic sense. Fagin et al. [17] improved on this result and introduced instance-optimal algorithms for the case when all sources provide either both sorted and random access, or only sorted access. Many variations of Fagin's original algorithm exist in the literature (see, e.g., [25, 20, 9]), for optimizing the execution of top-k queries over multimedia repositories. In particular, Güntzer et al. [20] reduce the number of random accesses through the introduction of more stop-conditions and exploitation of data distribution, and Chaudhuri and Gravano [9] propose a cost-based approach for query optimization, based on translating a given top-k query into a selection query that returns a superset of the actual top-k tuples.

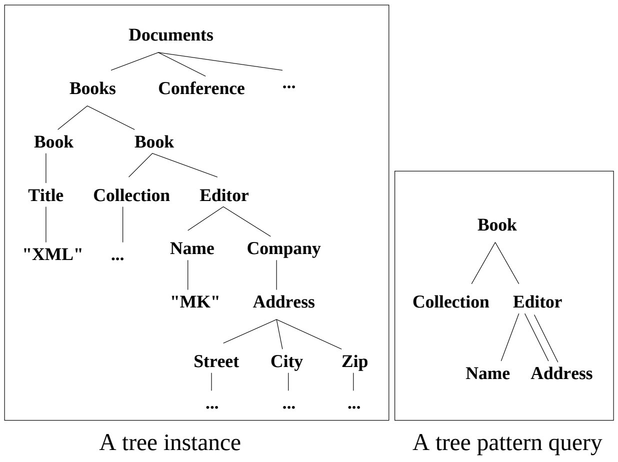
Over relational databases, Carey and Kossman [6, 7] present techniques to optimize top-k queries when the scoring is done through a traditional SQL order-by clause. If the scoring function involves multiple attributes, then this technique generally requires an initial scan of the complete relation during query processing. Donjerkovic and Ramakrishnan [15] propose a probabilistic approach to top-k query optimization, focusing on relations that might be the result of complex queries involving joins, for example, and where the ranking condition involves a single attribute. Finally, Chaudhuri and Gravano [10] exploit multidimensional histograms to process top-k queries by mapping them into traditional selection queries.

3 Problem Overview

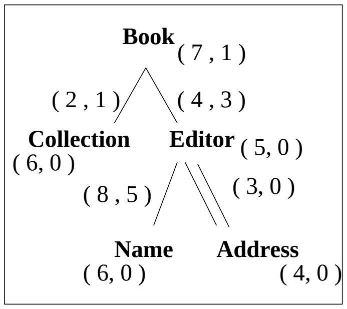
3.1 Background: Data Model and Queries

We consider a data model where information is represented as a forest of trees. Each node in the tree has an associated type. Types can be organized in a simple inheritance hierarchy. A simple example instance is given in Figure 1(a).

To abstract from existing query languages for XML (e.g. [8, 14, 22]), we express queries as tree patterns where nodes are types and edges are parent-child and ancestor-descendant relationships. These patterns are used to retrieve relevant portions of the data. Tree patterns do not capture all aspects of XML query languages such as ordering and restructuring. However, they form a key



(a) An Instance and a Query Example



(b) A Weighted query Example

Figure 1: Examples of an Instance and a Query

component of these query languages by focusing on their structural aspect. Figure 1(a) shows a concrete query example that looks for books that have a (child) editor, a (child) name, a (descendant) address and belong to a (child) collection. A single edge represents a parent-child relationship. A double edge is an ancestor-descendant relationship.

3.2 Relaxations

The main reason for approximate XML query matching is the heterogeneity of XML data which can make query formulation tedious. Users need to know well the content as well as the structure of the data to formulate queries which is not easy in the presence of optional elements in the data. Furthermore, users can themselves specify that, in addition to exact query matches, they expect "similar" answers to be returned (e.g., the "like" feature of amazon.com or the "advanced search" of google.com).

In order to explore the set of approximate matches of a query, we must be able to *relax* this query and guarantee that the set of answers that will be returned is a superset of the set of exact query matches.

One major difference between relaxing keyword-based searches and relaxing tree pattern queries is the use of structural predicates in tree queries (parent-child and ancestor-descendant relationships). One can think of several ways of relaxing a tree query. For example, if we are looking for books that appeared in a collection and that have an editor, we might want to return all books with an editor (including those that did not appear in a collection). To do so, we can make the element node `Collection` and the edge between this node and the `Book` node optional. Other kinds of relaxations modify conditions on nodes. For example, we could generalize the type `Book` to `Document`. In this paper, we made the choice of studying a few simple relaxations. For example, relaxations that increase the size of the original query by unfolding nodes are not studied.

3.3 Semantics

Definition 3.1 (Weighted Query) *A weighted query is a tree pattern where each node and edge is assigned up to two weights: an exact weight and (optionally) a relaxed weight. At a given node or edge, the exact weight is larger than the relaxed weight.*

Figure 1(b) shows an example of a weighted query corresponding to the tree query of Figure 1(a). We allow users to specify exact and relaxed weights at nodes and edges to provide maximal flexibility. However, in many cases, users have no particular preference. They might want to express that no one node (or edge) is more important than another in which case all nodes and/or all edges will have the same exact weight and the same relaxed weight.

Given a tree pattern query and an instance tree, we have the following definitions.

Definition 3.2 (Node Embedding) *A node embedding is a binding of a node in the query tree to a node in the instance tree such that the instance node is an instance of the type of the query node.*

Definition 3.3 (Edge Embedding) *An embedding of a child edge in the query is a binding of the edge to an edge in the instance tree where each endpoint of this edge is a node embedding. An embedding of a descendant edge in the query is a binding of the edge to a path in the instance tree where each endpoint of this edge is a node embedding.*

In other terms, a descendant edge in the tree query is mapped to a path in the instance tree. There can be several embeddings for a given query edge. By definition, each edge has a edge embedding which is empty (does not exist).

Definition 3.4 (Query Embedding) *A query embedding is an embedding of each node in the query.*

As an example, consider the query given in Figure 1(b) and suppose we are interested in the subquery containing `Book`, `Editor` and the edge between them. Suppose that the query node `Book` has two node embeddings b_1 and b_2 and `Editor` has one node embedding e_1. If there is an edge embedding between b_1 and e_1, then b_1 and e_1 along with the corresponding edge embedding correspond to a query embedding. If there does not exist an editor for book b_2, then there does not exist a query embedding that contains b_2.

Definition 3.5 (Exact Match) *Exact matches of a given query are all query embeddings.*

Definition 3.6 (Node Generalization) *We say that a node is generalized in a tree query when one of its super-types (in a type hierarchy) is used in the query instead of its type. A generalized node is also called a relaxed node.*

Definition 3.7 (Edge Relaxation) *We say that a child edge is relaxed in a tree query when it is transformed to a descendant edge. Descendant edges cannot be relaxed.*

Definition 3.8 (Optional Leaf Node) *We say that a leaf node is optional in a tree query when the node and the edge to its parent are removed from the tree query.*

This operation can be composed to make a whole subtree optional. However, in order to avoid empty queries, we forbid to make the root node optional.

Definition 3.9 (Subtree Promotion) *We say that a subtree is promoted when the edge between this the root of this subtree and its parent is removed and a descendant edge is created between this subtree and the parent node of its parent.*

Definition 3.10 (Relaxed Query) *A relaxed query is a query obtained by applying any composition of the four relaxations to it: (i) generalize a node, (ii) relax an edge, (iii) make a leaf node optional and, (iv) promote a subtree.*

The set of queries obtained by composing zero or more relaxations is the set of all possible relaxed queries of a given query. We call this set $\mathcal{R}$. By definition, the initial query is also included in $\mathcal{R}$.

Definition 3.11 (Approximate Match) *An approximate match is an answer that is an exact match to a relaxed query.*

By definition, an exact match is also an approximate match. Therefore, the set of all approximate matches to a query is obtained by computing the union of all exact matches of the relaxed queries in $\mathcal{R}$.

In order to produce a ranked list of query matches, the score of each match must be computed.

Definition 3.12 (Score of an Exact Node Match) *The score of an exact node match is the exact weight associated to the node to which it is matched in the weighted query.*

Definition 3.13 (Score of a Relaxed Node Match) *The score of a relaxed node match is no higher than the exact weight associated to the node to which it is matched in the weighted query and no smaller than its relaxed weight.*

There are different ways of computing the score of a relaxed node match. One could decide that if the most specific type of the node match is that of the corresponding node in the weighted query, then its score will be the exact weight of that node and if the most specific type of the match is a super-type of the original type, then its score is the relaxed weight of the original node. A more accurate way of computing the score of a relaxed node match is by using a function that depends on "how far", in the type hierarchy, the most specific type of a match is from the original type. An example is given below in a similar situation when computing the score of a relaxed edge match.

Definition 3.14 (Score of an Exact Edge Match) *The score of an exact edge match is the exact weight associated to the edge to which it is matched in the weighted query.*

Definition 3.15 (Score of a Relaxed Edge Match) *The score of a relaxed edge match, between a node n_1 and its descendant node n_2, is no higher than the exact weight of the edge between n_1 and n_2 and no smaller than its relaxed weight.*

There are different ways of computing the score of a relaxed edge match. One could decide that no matter how "far" node n_2 is from its ancestor node n_1 in the instance document, the score of the edge relating them will be the relaxed weight of the edge. In a more precise formulation, the "distance" between the two node matches could be taken into account in computing the score of the relaxed edge. A linear function in the distance or a logarithmic function could be used. The only constraint on that function is that it has to be monotonic. In other words, the further n_2 is from n_1, the smaller the score of the edge relating them should be. As an example, consider a query that looks for all instances where a Book has an Editor as a child. Suppose that we decide we can relax the child edge between Book and Editor into a descendant edge. Answers that are books with an editor as a descendant should have a lower score than those with an editor as a child (exact matches).

Assume the exact weight of the child edge between Book and Editor is *ew* and the relaxed weight is *rw*. Let *wdiff* = *ew*−*rw*, and *invldiff* = *1/(level(*Editor*)−level(*Book*))*. The score assigned to the edge between Book and Editor in an answer is computed using the formula: *ew*−*wdiff*∗ *(1−invldiff)*. For answers where Book and Editor appear in a parent/child relationship, *invldiff* is always 1 and the score of the match would be *ew* (the exact edge weight). Only when the level difference is infinite, would the score be *rw*. For a 2−level difference, *invldiff=1/2*, and the score is *(ew−(wdiff/2))*.

The same formula could be used to compute the score of a relaxed node match where the level difference measures how close to the original node type is the type of an actual match.

Our approach of evaluating relaxed tree patterns does not depend on the existence of a function to compute the scores. In fact, as a default, we can adopt the simplest solution of using exact weights for exact matches and relaxed weights for any relaxed match. If a function is given, it will be used.

Definition 3.16 (Score of a Match Against a (Relaxed) Query) *The score of a match against a query is the sum of the scores of all exact and relaxed nodes and edges in the match.*

As an example, the score of exact matches of the query given in Figure 1(b) is equal to the sum of the exact weights of its nodes and edges which is 45. If Book is generalized to Document, the score of exact matches (books) remains the same. However, the score of answers that match documents and not books is the sum of the relaxed weight of Book and the exact weights of the other nodes and edges in the weighted query 39=45-7+1.

Node that if a node is made optional (e.g. Address), the score of exact matches of a query can be 55 for those answers in which the editor has an address or 48 for those answers where the editor's address does not exist. When making a node optional, there is a *score-loss* equal 7=4+3 which corresponds to the fact that both the Address node and the edge between this node and its parent in the query do not have a value in some query matches.

A relaxed query might match the same answer multiple times with different scores. If we consider again the example in Figure 1(b) and if we suppose that Book is generalized to Document, the same book will be matched twice, when Book is considered and when Document is considered. In this case, the same answer will have a different score. In general, a match can be returned more than once with different score. Therefore, we define the score of an approximate match as follows:

Definition 3.17 (Score of an Approximate Match) *The score of an approximate match is the maximum score among all scores computed for it.*

Another example of the same query match appearing several times in the set of all query matches is the case where a subtree is promoted. Consider again the same example as before and suppose that the node Name is promoted to become a descendant node of Book. For the same book, editor and name there will be two approximate matches: the one in which the name is a child of editor and the one in which the name is a descendant of book. The two query embeddings have different scores and taking the one with the highest score (the one in which name is a child of editor) eliminates the other answer. Effectively, the set of approximate matches to a query has to be pruned to compute the correct set of query matches.

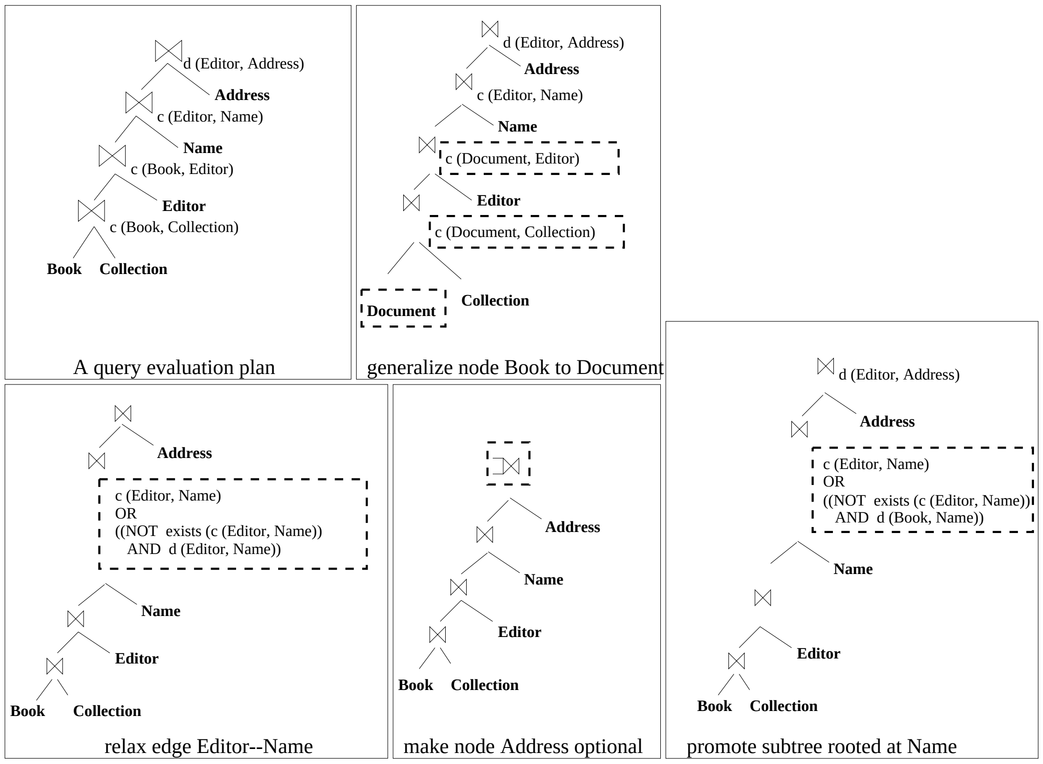

d (Editor, Address)
Address
c (Editor, Name)
Name
c (Book, Editor)
Editor
c (Book, Collection)
Book
Collection
A query evaluation plan
c (Document, Editor)
c (Document, Collection)
Document
generalize node Book to Document
relax edge Editor--Name
c (Editor, Name) OR ((NOT exists (c (Editor, Name)) AND d (Editor, Name))
make node Address optional
promote subtree rooted at Name
c (Editor, Name) OR ((NOT exists (c (Editor, Name)) AND d (Book, Name))

Figure 2: Relaxations

Definition 3.18 (Threshold Problem) *Given a query Q and a threshold t, the problem of finding the set of approximate matches of Q whose scores are $\geq t$ is the threshold problem.*

Definition 3.19 (Top-K Problem) *Given a query Q and an integer k, the problem of finding a set of k approximate matches of Q whose scores are greater than or equal to the kth highest query answer score is the top-k problem.*

4 Query Evaluation Plan and Relaxations Encoding

4.1 Query Evaluation Plan

Several query evaluation strategies have been proposed for XML (e.g. [22, 33]). They typically rely on a combination of index retrieval and join algorithms using specific structural predicates. For our study, we will use the join algorithm of [33] and two specific 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.

Figure 2 shows a translation of the tree query of Figure 1(a) into a left-deep join evaluation plan with the appropriate predicates. It is important to note that the relaxation techniques that we developed rely on the use of join plans to evaluate tree queries. However, they are not limited to a particular join evaluation algorithm. Further, we are not concerned with cost-based optimizations such as finding the best evaluation plan or join ordering. These issues are complementary to our relaxation techniques.

According to the query plan in Figure 2, an answer to a query is a tuple containing a value (possibly empty) for every leaf node in the query evaluation plan.

4.2 Encoding Relaxations in a Query Plan

We show how relaxations are encoded in the query evaluation plan. We also give examples to explain how answer scores vary with a varying number of relaxations. Figure 2 shows some relaxations of the query in Figure 1(a), and how to translate each of them to its corresponding expression in the evaluation plan. The modifications to the initial evaluation plan are highlighted with bold dashed lines. Predicates that are irrelevant to our discussion are omitted.

Generalizing nodes. In order to encode a node type generalization in a query plan, the predicate on the node type is replaced by a predicate on its super-type. We show in Figure 2 how `Book` can be replaced by `Document`.

Relaxing the structural predicate between nodes. In order to capture the relaxation of a child edge to a descendant edge, we transform predicates of the form $c(\tau_1, \tau_2)$ where τ_1 and τ_2 are two node types, into a predicate of the form ($c(\tau_1, \tau_2)$ OR ($\nexists$ $c(\tau_1, \tau_2)$ AND $d(\tau_1, \tau_2)$)). This new predicate first checks if a child relationship exists between the two nodes and then, if this condition is not satisfied, it checks if a descendant relationship exists between the two nodes. As an example, we relax the edge between `Editor` and `Name` and show the modified join condition in Figure 2. In the following figures, the predicate ($c(\tau_1, \tau_2)$ OR ($\nexists$ $c(\tau_1, \tau_2)$ AND $d(\tau_1, \tau_2)$) is simplified to ($c(\tau_1, \tau_2)$ OR $d(\tau_1, \tau_2)$), where OR has an ordered interpretation (check $c(\tau_1, \tau_2)$ first, $d(\tau_1, \tau_2)$ next).

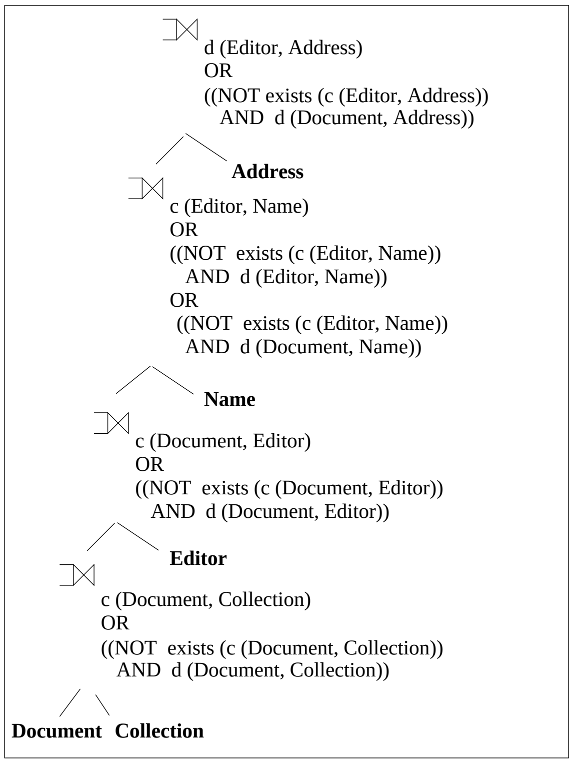
Making a leaf node optional. This operation causes the join that relates the leaf node to its parent node in the query evaluation plan to become an outer join. More precisely, it is a left outer join because we consider left-deep evaluation plans. An example of this operation is given in Figure 2 where `Address` is made optional. The outer join guarantees that even books whose editor does not have an address will be returned.

Promoting a subtree. This operation causes the subtree to become a descendant of the parent of its current parent. In the query evaluation plan, the predicate between the parent of the subtree and the root of this subtree is modified. Consider the example in Figure 2. Promoting `Name` causes the transformation of the predicate that relates `Name` to its parent node `Editor` to be modified. If the predicate between the root of the promoted subtree (here `Name`) and its parent (here `Editor`) is of the form: `c(Editor,Name)`, it becomes: (`c(Editor,Name)` OR ($\nexists$(`c(Editor,Name)`) `d(Book,Name)`). This new predicate checks for a descendant relationship between `Book` and `Name`. In the following figures, the predicate ($c(\tau_1, \tau_2)$ OR ($\nexists$ $c(\tau_1, \tau_2)$ AND $d(\tau_3, \tau_2)$) is simplified to ($c(\tau_1, \tau_2)$ OR $d(\tau_3, \tau_2)$), where OR has an ordered interpretation (check $c(\tau_1, \tau_2)$ first, $d(\tau_3, \tau_2)$ next).

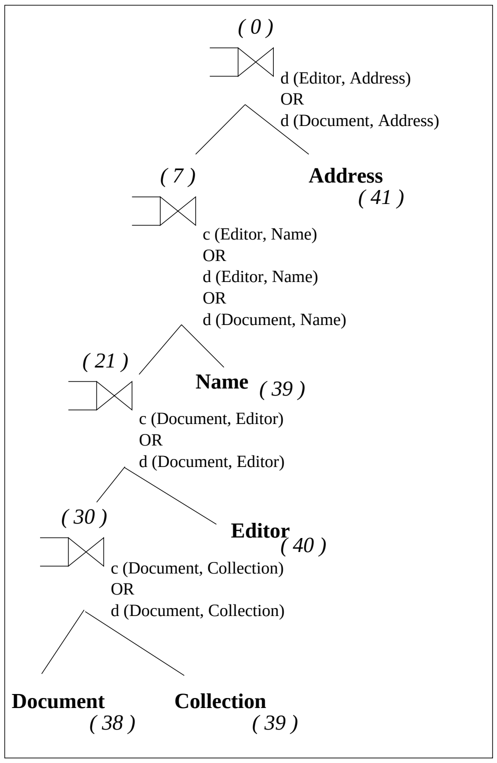
Combining Relaxations. Figure 3(a) shows an example of combining all possible relaxations in the same query. Each node is generalized if a type hierarchy exists (in our example only `Book` becomes `Document`). Therefore, all predicates where `Book` was involved now have `Document` as an argument. All child edges are relaxed except the one between `Editor` and `Address` (since it was already a descendant edge). All nodes, except the root, are made optional and all subtrees are promoted. Note that by composing relaxations, `Editor` becomes a leaf node at some point and can thus be made optional (see the outer join between `Document` and `Editor`). Therefore, in order to make a non-leaf node optional, all its subtrees must have been promoted.

5 Threshold Approach

The goal of the threshold approach is to take a weighted tree query and a threshold and generate, in a decreasing order of answer scores, a ranked list of answers whose scores are at least as large as the threshold. There is a whole body of work in IR which aims to evaluate answers to keyword-based searches and compute scores that measure the relevance of an answer to a set of keywords [28]. A naive algorithm one could think of in the case of tree queries, is to translate a query into an evaluation plan, encode all possible relaxations, evaluate answers to the relaxed query along with the score of each answer and finally, keep only 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 build the set of answers and only then prune irrelevant ones. In order to evaluate relaxed queries more efficiently, we need to detect, as soon as possible in the query evaluation process, which intermediate answers will never meet the threshold. To do so, we designed the algorithm `Thres`. `Thres` operates on a join plan. Before describing this algorithm, we explain how answer scores are computed at each step of the join plan.


d (Editor, Address)
OR
((NOT exists (c (Editor, Address))
AND d (Document, Address))
Address
c (Editor, Name)
OR
((NOT exists (c (Editor, Name))
AND d (Editor, Name))
OR
((NOT exists (c (Editor, Name))
AND d (Document, Name))
Name
c (Document, Editor)
OR
((NOT exists (c (Document, Editor))
AND d (Document, Editor))
Editor
c (Document, Collection)
OR
((NOT exists (c (Document, Collection))
AND d (Document, Collection))
Document
Collection

(a) Relaxations



(b) A `Thres` Example

Figure 3: Relaxations and `Thres`

5.1 Computing Answer Scores

The following query is general enough to show most cases that might occur in an evaluation plan: $Q =$ `Document` $\sqsupset\!\!\!\bowtie_{c(Document,Collection)\ OR\ d(Document,Collection)}$ `Collection`.

Suppose that the node `Document` is in fact, a relaxation of an initial node `Book`. Evaluating `Document` results in two kinds of answers: (1) answers whose type is the exact node type `Book` and (2) answers whose type is the relaxed node type `Document` but not `Book`. Answers in the first category are assigned the exact weight of this node. Answers in the second category are assigned a smaller weight.

Given two intermediate answers *doc* (of type `Document`, with score s_1) and *coll* (of type `Collection`, with score s_2), the result of joining them can be one of three kinds:

- *doc* answers that do not join with any *coll* answer. These answers are assigned s_1 as a score.
- *doc* answers that join with *coll* answers via the predicate c(`Document`,`Collection`).
- Finally, *doc* answers that do not join with *coll* answers via c(`Document`,`Collection`) but that join with *coll* answers via the predicate d(`Document`,`Collection`).

Answers that result from a join (via the child or the descendant predicate) are assigned the score: $s_1 + s_2 + s$(`Document, Collection`) where s(`Document, Collection`) is the weight of the edge between *doc* and *coll* which depends on the level difference between the actual instances of `Document` and `Collection` (see Section 3.3 for more details).

5.2 `Thres` Algorithm

Consider the example in Figure 3(b). The query pattern is first translated into a join evaluation plan and all possible relaxations are encoded.

Definition 5.1 (Maximal Weight) *Maximal Weight,* `maxW`*, is defined as the maximal weight by which intermediate answers in a query plan are expected to grow.*

`maxW` is the total weight of what remains to be computed at a given step of the evaluation plan. When no intermediate query answer is computed, `maxW` is the sum of all the exact weights of nodes and edges in the query tree. In this case, `maxW` is also the maximal score a query match can have. `maxW` can be computed at each node of a query evaluation plan. For example, in Figure 3(b), `maxW` of the `Document` node is `38`. This number is obtained by computing the sum of the exact weights of the nodes and edges of the subtree rooted at `Document` (excluding the exact weight of `Document` itself). By definition, `maxW` of the last join, the one which is the root of the evaluation plan, is `0` since answers to the query are totally computed at the root node.

`Thres` is summarized in Figure 4. Initially, it computes `maxW` at each node of the evaluation plan. The query is evaluated in a bottom-up fashion. At each node, there is a set of intermediate query results along with their scores. If the sum of the score of an answer and `maxW` at the node does not meet the threshold, this intermediate result is eliminated. Note that Figure 4 shows a nested loop evaluation but this is only for the purpose of understanding. The algorithms we use for joins and left-outer joins are based on the MPMGJN algorithm of [33].

5.3 Adaptive Query Optimization

The optimizations we are focusing on in this section rely on a simple cost model according to which the larger is the size of the output of a join operation, for a given join algorithm, the higher is the cost of this join. Based on this assumption, we can draw a simple cost lattice in which a left outer join is always more expensive than a join (since it potentially produces more answers) and a join

```
evaluateSubtree(Node n)
   if (n is leaf) results=evaluateLeaf(n);
      for (r in results)
         if (r->score+n->maxW < threshold)    results=results-r;
      return results;
   list1 = evaluateSubtree(n->left);
   list2 = evaluateSubtree(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,s,n->results);
   return results;
```

Figure 4: `Thres` Algorithm

(or outer join) with a descendant predicate is more expensive than a join (or outer join) with a child predicate (because children are also descendants). Relevant answers are those whose scores are at least as large as a given threshold. Since a threshold is always between the minimal score and the maximal score of an answer, irrelevant answers are those obtained by "relaxing the query too much". Therefore, we want to optimize a query plan by detecting unnecessary relaxations (those that will always generate irrelevant answers) and undo them. This detection can be done statically or adaptively. The static detection is obvious. For example, it can be applied to decide whether a node can be made optional or not. If the sum of the exact weights of all nodes and edges in the query, except that of the node itself and that of the edge relating it to its parent, is smaller than the threshold, then this node should not be made optional. In the following, we will focus on the description of our adaptive optimization, `OptiThres`.

5.3.1 `OptiThres` Algorithm

The idea behind `OptiThres`, an improved version of `Thres`, is that we can predict, during query evaluation, if a relaxation produces additional matches that will meet the threshold. We know that relaxing a query causes some results to have a lower score. By keeping enough information in the query evaluation process, we can dynamically decide whether applying a relaxation generates answers that will never meet the threshold in which case, we undo this relaxation in the query. `OptiThres` cannot be done statically since it depends on the scores of intermediate answers.

The algorithm `OptiThres` is given in Figure 5. Only the parts that are additions to `Thres` are specified. When necessary, we indicate which part of `Thres` is missing.

Let us take a simple example. Consider the query pattern in Figure 6. This query looks for all `Proceedings` that have a `Publisher` and a `Month`. Exact and relaxed weights are associated with each node and edge in the query. `Proceeding` is relaxed to `Document`, `Publisher` is relaxed to `Person` and the child edges are relaxed to descendant ones. `Person` is made optional. The threshold is set to `14`. Statically, we cannot make any decision about forbidding relaxations.

While `Thres` relies on `maxW` at each node, `OptiThres` computes three weights at each join node of the query evaluation plan (e.g. at the first join (`11, 8, 9`) are computed). The first weight, `relaxNode`, is used to decide whether the right child of the join node (that will be joined with the left child of the join node) should remain relaxed (assuming it was relaxed). `relaxNode` is obtained

```
evaluateSubtree(Node n)
   if (n is leaf) results=evaluateLeaf(n);
   /* maxLeft is now set to the maximal score of the results in n */
      if (maxLeft+relaxNode < threshold)    unrelax(n->right);
   // Thres part that prunes leaf nodes is dropped.
      return(results);
   list1 = evaluateSubtree(n->left);
   list2 = evaluateSubtree(n->right);
   /* both maxLeft and maxRight are set */
   if (maxLeft+relaxJoin < threshold)    unrelax(n->join);
   if (maxLeft+maxRight+relaxPred < threshold)    unrelax(n->join->predicate);
   // Thres part that prunes join results
   return results;
```

Figure 5: `OptiThres` Algorithm

by computing the sum of the exact weights of all nodes and edges that remain to be computed and the highest relaxed weight of the right child of the join node. In this example, we assume that we are using the same relaxed weight for all approximate matches that are not exact matches (no function is used). Therefore, a person who is a publisher will be assigned score `10` and a person who is not a publisher will be assigned `1`. Therefore, `relaxNode` is equal to `(2+6+2+1)` where 1 is the highest relaxed weight a person can have. The second weight, `relaxJoin`, is used to decide whether the current join should remain an outer join or should be replaced by an inner join. It is obtained by computing the sum of the exact weights of all the remaining nodes and edges except the right child of the join itself `(6+2)`. Finally, the third weight, `relaxPred`, is used in case the join has been modified (from an outer join to a join) and will help decide if the predicate at this join should remain relaxed. It is obtained by computing the sum of the exact weights of all remaining nodes and edges and the highest relaxed weight of the predicate. In the same manner as for `relaxNode`, we assume that the highest relaxed weight of the predicate is `1` which corresponds to the relaxed weight of the edge between `Proceeding` and `Publisher`. `relaxPred` is then `(6+2+1)`. The decision of whether to undo a predicate relaxation (turn it back to a child predicate) is done only if one has decided to turn a join operation from an outer join to a join. Intuitively, this means that if we are willing to "lose" a whole subtree by keeping an outer join, we do not need to check for the predicate relaxation.

Let us look in more details at the example in Figure 6 with a threshold set to `14`. `Document` is evaluated first. Assume that the maximal score in the list of answers that we get is `2` which means there are no `Proceedings` in our database. At the next join, `relaxNode` is equal to `11`, `relaxJoin` is `8` and `relaxPred` is `9`. The sum `relaxNode+2=13` is smaller than the threshold. In this case, we decide *not* to relax `Publisher` to `Person` and the query is modified consequently. We now evaluate `Publisher`. The maximal score of the result list is `10`. We compute `relaxJoin+2=10` and decide that we need to turn the outer join into a join since the threshold is `14` and we cannot "afford losing `Publisher`". We now check whether we should keep the descendant predicate. We use `relaxPred: 2+10+relaxPred=21` which is higher than the threshold. Therefore, we keep the descendant predicate at the join. During the join evaluation, `Thres` can be applied to prune the join result. Once this is done, suppose that the maximal score of its output is `10+2+2=14`. At this

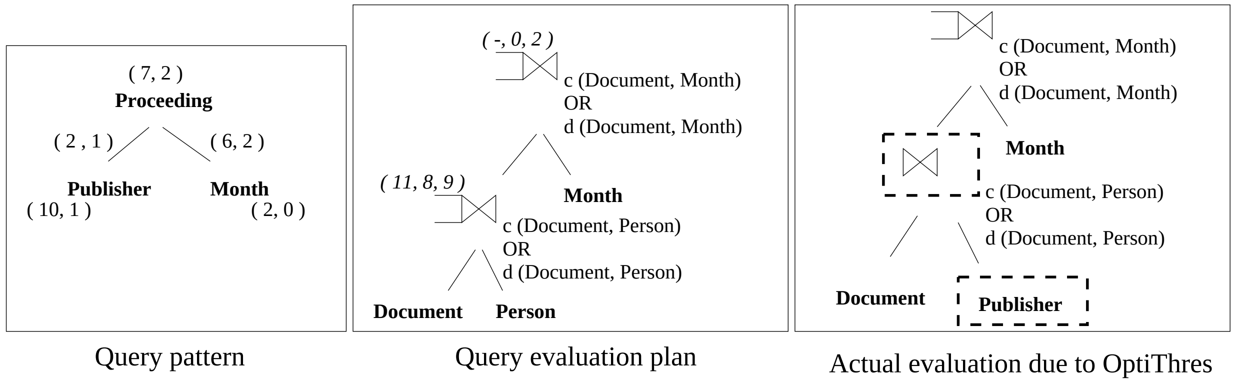


Figure 6: A Simple `OptiThres` Example

point, we do not consider whether we should undo any relaxation at the leaf node `Month` since it has not been generalized (`relaxNode` at the second join is not specified). We then evaluate `Month` and get answers with a score of `2`. We use the current `relaxJoin` value `0` to decide whether we should turn the outer join to a join. Effectively, `14+0=14` does meet the threshold which means that we can keep the outer join as it is. Therefore, we do not use `relaxPred` at this point. We now compute the join and apply `Thres`. The expression that we have effectively computed is given in Figure 6 and the dynamically modified parts in the query are highlighted.

5.3.2 Properties of the Adaptive Optimization Algorithm

Complexity : Since `OptiThres` makes use of the maximal score of answers at each step of the evaluation process and some precomputed numbers at each node in the evaluation tree, the complexity added to the joins depends only on the size of the query evaluation plan. It does not depend on the data. Hence, `OptiThres` is linear in the size of the tree pattern.

Comparison with `Thres`: It must be noted that even if `OptiThres` can have a potentially high efficiency benefit over `Thres`, it is used only to decide whether some relaxations should be undone and is not enough for detecting which answers should be pruned. `Thres` remains necessary.

However, since the only decision that `Thres` makes on leaf nodes of the evaluation plan is whether they should be generalized or not, if `OptiThres` is applied on these nodes, `Thres` does not need to be applied on them. The only leaf node of the evaluation plan where `OptiThres` cannot be used to decide whether it should be generalized or not is the root of the query tree since it is the first node that is evaluated. However, this decision is made statically.

At each step of the query evaluation process, `Thres` and `OptiThres` will produce the same output size. The difference between these two algorithms lies in the size of the non-pruned output. While `Thres` pruning might work on a larger set of outputs, `OptiThres` might decide, in advance that a relaxation should not be applied and thus produce a smaller non-pruned set.

Locality Property : `OptiThres` has an important locality property that guarantees its low (linear) complexity. No global decision about "undoing" relaxations is necessary. At each join node, there is enough information to decide to undo relaxations local to this node (not generalizing the right child of the join, turning an outer join to a join or turning a descendant

```
evaluateSubtree(Node n)
   if (n is leaf) results=evaluateLeaf(n);
      for (r in results)
         if (r->score+n->maxW < prunScore)    results=results-r;
      return results;
   list1 = evaluateSubtree(n->left);
   list2 = evaluateSubtree(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 >= prunScore)    append(r1,r2,s,n->results);
prunScore = determinePrunScore(hash);
   return results;
```

Figure 7: `Top-K` Algorithm

predicate to a child one). Making these decisions at nodes other than the current one (in a non-local fashion) will not result in a better choice. In other words, applying `OptiThres` locally at each node is at least as good as applying it globally since a global optimization would rely on more conservative estimates.

6 Top-K Approach

`Top-K` works in the same spirit as `Thres`. The algorithm is given in Figure 7. The only difference with `Thres` is that instead of having a *fixed* threshold, it relies on a dynamically computed one, `prunScore` which has the value of the score of the kth answer at each step. `prunScore` is used at each step of the query evaluation process (at each left-outer join operation) to prune answers that will never participate in the final result. The idea behind this algorithm is simple. At each step, intermediate answers are ranked by their score and the score of the kth answer, `prunScore`, is used as the "current threshold" and is considered for pruning purposes. At a given step (a given left-outer join node), the score of an intermediate answer, augmented with `maxW` at this node represents the highest score this answer will have. If this highest score is smaller than `prunScore` at this node, it means that this intermediate answer will never be in the top-k ones and can be eliminated at this step. Note that this reasoning is possible because all the join operations used in our query evaluation plan are left outer joins that guarantee that the current set of answers will always be selected.

`Top-K` can be seen as a variant of `Thres` where the threshold value is not known in advance and changes dynamically at each join evaluation. We will see experimentally that if the threshold corresponding to query answers is estimated in advance, `Thres` can be used to retrieve all answers whose scores are at least as large as that threshold and thus, all top-k answers.

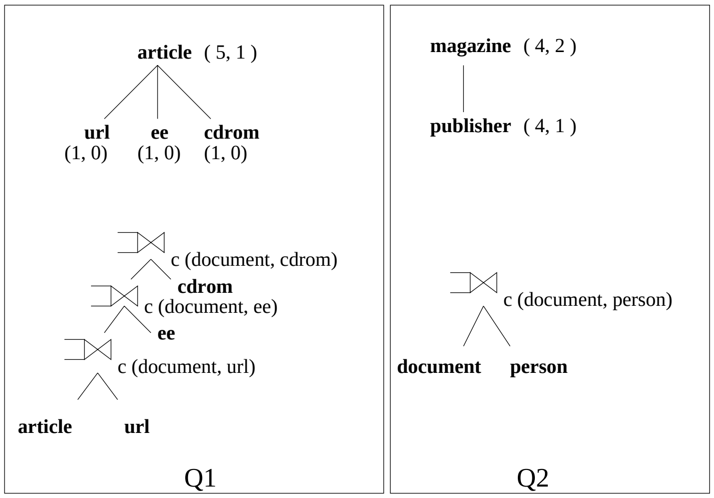


Figure 8: Queries Used in the Experiments

7 Experiments

7.1 Experimental Setup

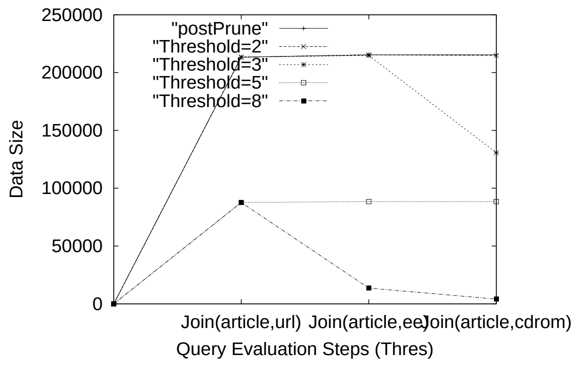
We use the `DBLP XML` dataset which is approximately 85MBytes and contain 2.1M elements. More details on the data size are in the table below. The `DTD` of this dataset as well as the data itself can be found at *http://dblp.uni-trier.de/db*. In order to prune data at each step of query evaluation, we modified the join algorithm of [33] so that each input (and output) is materialized in a file. We run all our experiments on a HP-UX machine with 32MBytes of memory.

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

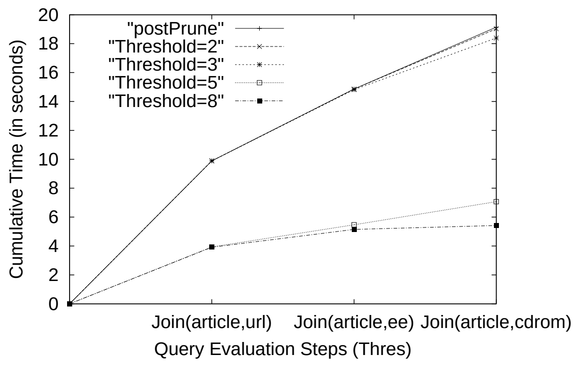
We define a type hierarchy where `document` is a super-type of `book`, `incollection`, `inproceedings`, `proceedings`, `article`, `phdthesis`, `mastersthesis`, `www` and `magazine`; and `person` is a super-type of `author`, `editor` and `publisher`. We use the queries of Figure 8. Since the DTD does not have long paths, we do not use edge relaxation. In all experiments, query time is reported in seconds and data size in the number of answers.

7.2 Studying Thres

We use **Q1** where `url`, `ee`, and `cdrom` are made optional and `article` is relaxed to `document`. We compare the time it takes to perform `Thres` (with multiple threshold times) and `postPrune` (post-pruning) and also the cumulative size of the data that is processed by each algorithm. The results are given in Figure 9 where the x-axis represents each step of **Q1** evaluation. Figure 9(a) shows that the higher the threshold, the earlier irrelevant data is pruned and evaluation time is reduced because the amount of data that remains in the evaluation process is reduced (as shown in Figure 9(b)). For `postPrune`, data pruning occurs only at the last step of query evaluation.



(a) Data Size Comparison



(b) Time Comparison

Figure 9: Compare `Thres` and `postPrune`

7.3 Benefit of OptiThres

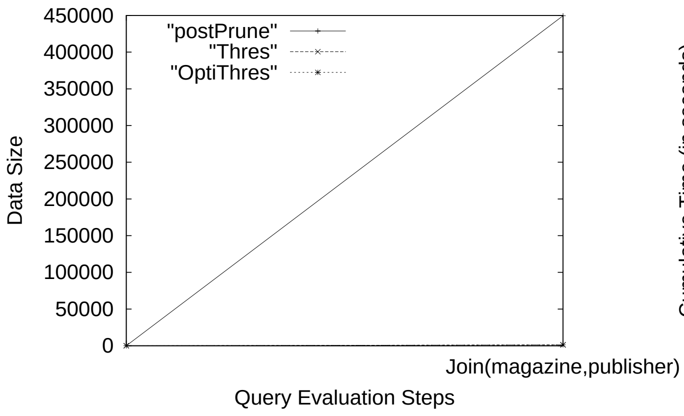
We compare `postPrune`, `Thres` and `OptiThres`. We use query **Q2** with a threshold equal to 5 because we want to show how `OptiThres` decides that `publisher` should not be relaxed to `person`. First, `magazine` is relaxed to `document` and `publisher` to `person`. `person` is made optional. Figure 10 shows a size and a time comparison. For a fair comparison, the same set of answers (with the same size) is produced by all our algorithms. `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 a join). This is because `magazine` is empty and the only instances that are selected when evaluating `magazine` are those that are documents. Thus their score is 2 and do not meet the threshold if `publisher` is relaxed. The graphs of Figure 10(a) show that both `postPrune` and `Thres` scan all of `person` which results in processing a higher data size than `OptiThres` which scans only `publisher`. This also results in a higher execution time as shown in Figure 10(b). In addition, since `OptiThres` performs a join operation (instead of an outer join), there are time and data size savings at the last step of query evaluation.

Finally, since `OptiThres` prunes data earlier than the other strategies, the amount of data that it manipulates is smaller and thus its execution time is the smallest. Due to its ability to undo unnecessary relaxations, `OptiThres` achieves a significant improvement in query evaluation performance. In addition, since `OptiThres` does not depend on the data and is an in-memory check, it does not generate any additional I/O costs and is very efficient.

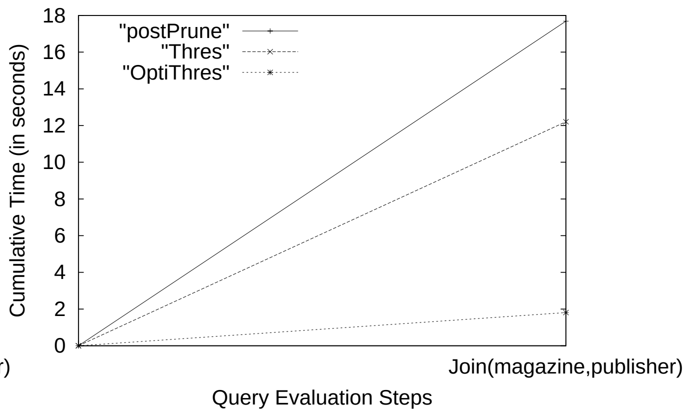
7.4 Top-K

We study the performance of `Top-K` and compare it to `Thres` and `postPrune`. We run two sets of experiments using `Q1` with two different weights. In the first case, `Q1` has the same weights as in Figure 8. In the second case, `article` has weights `(15,3)`, `cdrom` has weight `5`, `ee` has weight `6` and `month` has weight `2`. With this choice of weights, different combinations of relaxations result in different scores of matching answers. The purpose of this experiment is to show that running `Thres` with a very precise threshold value is always faster than running `Top-K` with the corresponding k value.

The graphs in Figure 11 illustrate the results of the first experiment. We run this experiment



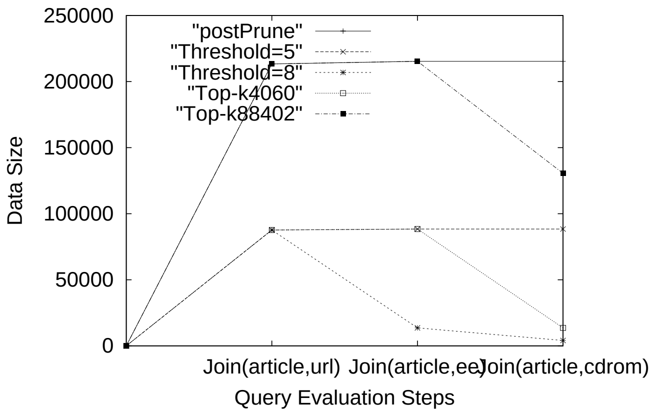
(a) Data Size Comparison



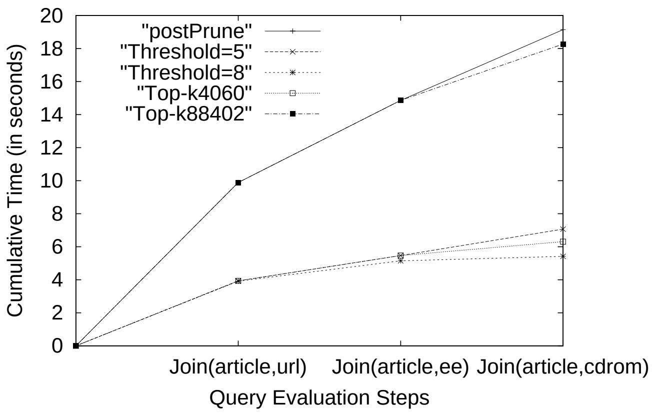
(b) Time Comparison

Figure 10: Compare `OptiThres, Thres` and `postPrune`

with two threshold values. With each value, we count the exact number of answers and used that number as value for k. For example, in the case where the threshold is `8`, k is `4060`. In the case where the threshold is set to `8`, `Top-K` is almost as efficient as `Thres` because it detects at the first join operation that all answers where `article` has been relaxed to `document` will never be in the top-k answers and prunes them. However, with a threshold set to `5`, `Top-K` is worse because it does not prune irrelevant data early enough in the evaluation process. In the second experiment where `Q1` is considered with a different weight, `Top-K` always behaves worse than `Thres` with threshold values `28` and `20` but starts behaving almost as well as `Thres` with smaller threshold values. For space limitation reasons, we do not graphically show the results of this second experiment.



(a) Size



(b) Time

Figure 11: Compare `Top-K OptiThres` and `postPrune`

The conclusion of this experiment is that with a very precise threshold value `Thres` will always be faster than `Top-K`. However, this requires a very good estimation of the threshold value that corresponds to a given value of k. How to do so accurately is outside the scope of this paper.

7.5 Comparison with Rewriting-Based Approaches

We run all our algorithms (except `Top-K`) on query `Q1` with a threshold set to 2 (to select a high 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 time is obtained by computing the sum of executing each query. `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 faster than the other strategies. `OptiThres` is considerably faster than rewriting-based approaches. The reason is that `MultiQ` performs 10 joins, `MultiQOptim` performs 8 joins and `OptiThres` performs 3 joins.

8 Conclusion

In this paper we provided a general framework for relaxing tree queries and combining our relaxation algorithms with existing join algorithms to evaluate these kinds of queries. We defined query relaxation for tree patterns and presented efficient algorithms for the evaluation of relaxed tree patterns. We presented a data pruning strategy inspired by existing IR techniques and experimentally showed its benefit over post-pruning and rewriting-based approaches.

References

[1] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree Pattern Queries Minimization. SIGMOD'01.

[2] R. Baeza-Yates, G. Navarro. Proximal Nodes: A Model to Query Documents Databases by Content and Structure. ACM Transactions on Information Systems 15(4), pages 400-435.

[3] R. Baeza-Yates, G. Navarro. XQL and Proximal Nodes. ACM SIGIR 2000 Workshop on XML and Information Retrieval. July 28, 2000. Athens, Greece.

[4] R. Baeza-Yates, G. Navarro. Integrating contents and structure in text retrieval. ACM SIGMOD Record, March 1996. 25(1):67-79.

[5] E. W. Brown. Fast evaluation of structured queries for information retrieval. *Proc. ACM SIGIR Conf.*'95.

[6] M. J. Carey and D. Kossmann. On saying "Enough Already!" in SQL. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1997.

[7] M. J. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the International Conference on Very Large Databases*, 1998.

[8] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, M. Stefanescu XQuery 1.0: An XML Query Language. http://www.w3.org/TR/query-datamodel/.

[9] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1996.

[10] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *Proceedings of the International Conference on Very Large Databases*, 1999.

[11] Y. Chiaramella, P. Mulhem, 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.

[12] DBLP Database. http://www.informatik.uni-trier.de/ ley/db/index.html.

[13] 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). Viterbo, Italy.

[14] A. Deutch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. A Query Language for XML. WWW'99.

[15] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top N queries. In *Proceedings of the International Conference on Very Large Databases*, 1999.

[16] R. Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1996.

[17] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2001.

[18] C. Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1), 1985.

[19] N. Fuhr, K. Grossjohann. XIRQL – An Extension of XQL for Information Retrieval. To appear in ACM SIGIR 2001.

[20] U. Guntzer, W.-T. Balke, and W. Kiessling. Optimizing multi-feature queries for image databases. In *Proceedings of the International Conference on Very Large Databases*, 2000.

[21] Y. Hayashi, J. Tomita, G. Kikui. Searching Text-rich XML Documents with Relevance Ranking. ACM SIGIR 2000 Workshop on XML and Information Retrieval. July 28, 2000. Athens, Greece.

[22] J. McHugh, S Abiteboul, R. Goldman, D. Quass, J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record, 26(3):45-66, 1997.

[23] Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 2001.

[24] S. Myaeng, D.-H. Jang, M.-S. Kim, Z.-C. Zhoo. A flexible Model for Retrieval of SGML Documents. In [Croft et al. 98]. pages 138-145.

[25] S. Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the IEEE International Conference on Data Engineering*, 1999.

[26] M. Persin. Document filtering for fast ranking. *Proc. ACM SIGIR Conf.*, Dublin, Ireland, 1994.

[27] J. Robie, J. Lapp, and D. Schach. XML query language (XQL). Available from http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[28] G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, New York, 1983.

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

[30] A. Theobald, G. Weikum. Adding Relevance to XML. WebDB'00.

[31] H. Turtle and J. Flood. Query evaluation: Strategies and Optimization. *Information Processing & Management*, Nov. 1995.

[32] W. Y. P. Wong and D. L. Lee. Implementation of partial document ranking using inverted files. *Information Processing & Management*, 29(5), Oct. 1993.

[33] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD'01.

[34] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. VLDB'92.