CoXML: COOPERATIVE XML QUERY ANSWERING

INTRODUCTION

As the World Wide Web becomes a major means in disseminating and sharing information, there has been an exponential increase in the amount of data in web-compliant format such as HyperText Markup Language (HTML) and Extensible Markup Language (XML). XML is essentially a textual representation of the hierarchical (tree-like) data where a meaningful piece of data is bounded by matching starting and ending tags, such as `<name>` and `</name>`. As a result of the simplicity of XML as compared with SGML and the expressiveness of XML as compared with HTML, XML has become the most popular format for information representation and data exchange.

To cope with the tree-like structure in the XML model, many XML-specific query languages have been proposed (e.g., XPath[1] and XQuery (1)). All these query languages aim at the exact matching of query conditions. Answers are found when those XML documents match the given query condition *exactly*, which however, may not always be the case in the XML model. To remedy this condition, we propose a cooperative query answering framework that derives *approximate* answers by relaxing query conditions to *less* restricted forms. Query relaxation has been successfully used in relational databases (e.g., Refs. 2–6) and is important for the XML model because:

1. Unlike the relational model where users are given a relatively small-sized schema to ask queries, the schema in the XML model is substantially bigger and more complex. As a result, it is unrealistic for users to understand the full schema and to compose complex queries. Thus, it is desirable to relax the user's query when the original query yields null or not sufficient answers.
2. As the number of data sources available on the Web increases, it becomes more common to build systems where data are gathered from heterogeneous data sources. The structures of the participating data source may be different even though they use the same ontologies about the same contents. Therefore, the need to be able to query differently structured data sources becomes more important (e.g., (7,8)). Query relaxation allows a query to be structurally relaxed and routed to diverse data sources with different structures.

Query relaxation in the relational model focuses on value aspects. For example, for a relational query "*find a person with a salary range 50K–55K*," if there are no answers or insufficient results available, the query can be relaxed to "*find a person with a salary range 45K–60K*." In the XML model, in addition to the value relaxation, a new type of relaxation called *structure relaxation* is introduced, which relaxes the structure conditions in a query. Structure relaxation introduces new challenges to the query relaxation in the XML model.

FOUNDATION OF XML RELAXATION

XML Data Model

We model an XML document as an ordered, labeled tree in which each element is represented as a node and each element-to-subelement relationship is represented as an edge between the corresponding nodes. We represent each data node u as a triple (*id, label*, <*text*>), where *id* uniquely identifies the node, *label* is the name of the corresponding element or attribute, and *text* is the corresponding element's text content or attribute's value. *Text* is optional because not every element has a text content.

Figure 1 presents a sample XML data tree describing an article's information. Each circle represents a node with the node *id* inside the circle and *label* beside the circle. The text of each node is represented in italic at the leaf level.

Due to the hierarchical nature of the XML data model, we consider the text of a data node u as part of the text of any of u's ancestor nodes in the data tree. For example, in the sample XML data tree (Fig. 1), the node 8 is an ancestor of the node 9. Thus, the text of the node 9 (i.e., "*Algorithms for mining frequent itemsets...*") is considered part of the text of the node 8.

XML Query Model

A fundamental construct in most existing XML query languages is the tree-pattern query or *twig*, which selects elements or attributes with a tree-like structure. In this article, we use the twig as our basic query model. Similar to the tree representation of XML data, we model a query twig as a rooted tree. More specifically, a query twig T is a tuple (*root, V, E*), where

- *root* is the root node of the twig;
- *V* is the set of nodes in the twig, where each node is a tripe (*id, label,* <*cont*>), where *id* uniquely identifies the node, *label* is the name of the corresponding element or attribute, and *cont* is the content condition on the corresponding node. *cont* is optional because not every query node may have a content condition;
- The content condition for a query node is either a database-style value constraint (e.g., a Boolean condition such as equality, inequality, or range constraint) or an IR-style keyword search. An IR-style content condition consists of a set of terms, where

[*]This work is supported by NSF Award ITR#: 0219442

[1]See http://www.w3.org/TR/xpath/.

Wiley Encyclopedia of Computer Science and Engineering, edited by Benjamin Wah.

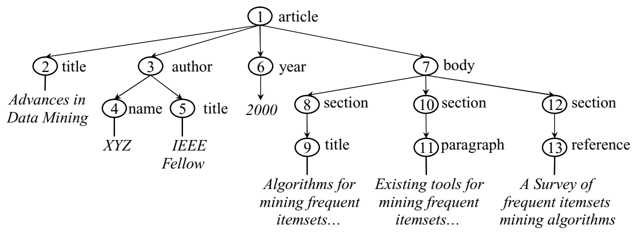


Figure 1. A sample XML data tree.

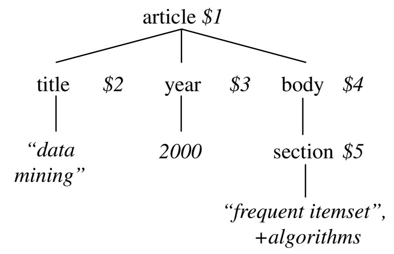


Figure 2. As sample XML twig.

each term is either a single word or a phrase. Each term may be prefixed with modifiers such as "+" or "−" for specifying preferences or rejections over the term. An IR-style content condition is to be processed in a nonBoolean style; and

- E is the set of edges in the twig. An edge from nodes $\$u$[2] to $\$v$, denoted as $e_{\$u,\$v}$, represents either a parent-to-child (i.e., "/") or an ancestor-to-descendant (i.e., "//") relationship between the nodes $\$u$ and $\$v$.

Given a twig T, we use $T.root$, $T.V$, and $T.E$ to represent its root, nodes, and edges, respectively. Given a node $\$v$ in the twig T (i.e., $v \in T.V$), we use $\$v.id$, $\$v.label$, and $\$v.cont$ to denote the unique ID, the name, and the content condition (if any) of the node respectively. The IDs of the nodes in a twig can be skipped when the labels of all the nodes are distinct.

For example, Fig. 2 illustrates a sample twig, which searches for articles with a title on "*data mining*," a year in 2000, and a body section about "*frequent itemset algorithms*." In this query, the user has a preference over the term *algorithm*. The twig consists of five nodes, where each node is associated with an unique *id* next to the node. The text under a twig node, shown in italic, is the content or value condition on the node.

The terms "twig" and "query tree" will be used interchangeably throughout this article.

XML Query Answer

With the introduction of XML data and query models, we shall now introduce the definition of an XML query answer. An answer for a query twig is a set of data nodes that satisfy the structure and content conditions in the twig. We formally define a query answer as follows:

Definition 1. ***Query Answer*** *Given an XML data tree D and a query twig T, an answer for the twig T, denoted as* $\mathcal{A}_D^T$*, is a set of nodes in the data D such that*:

- $\forall \$u \in T.V$, *there exists an unique data node u in* $\mathcal{A}_D^T$ *s.t* $\$u.label = u.label$. *Also, if* $\$u.cont \neq null$ *and* $\$u.cont$ *is a database-style value constraint, then the text of the data node* $u.text$ *satisfies the value constraint. If* $\$u{:}cont \neq null$ *and* $\$u.cont$ *is an IR-style content condition, then the text of* $u.text$ *should contain all the terms that are prefixed with "+" in* $\$u.cont$ *and must not contain any terms that are prefixed with "−" in* $\$u.cont$;
- $\forall e_{\$u,\$v} \in T.E$, *let u and v be the data nodes in* $\mathcal{A}_D^T$ *that correspond to the query node* $\$u$ *and* $\$v$, *respectively, then the structural relationship between u and v should satisfy the edge constraint* $e_{\$u,\$v}$.

For example, given the twig in Fig. 2, the set of nodes {1, 2, 6, 7, 8} in the sample XML data tree (Fig. 1) is an answer for the query, which matches the query nodes {\$1, \$2, \$3, \$4, \$5}, respectively. Similarly, the set of nodes {1, 2, 6, 7, 12} is also an answer to the sample query. Although the text of the data node 10 contain the phrase "*frequent itemset*," it does not contain the term *algorithm*, which is prefixed with "+." Thus, the set of data nodes {1, 2, 6, 7, 10} is not an answer for the twig.

XML Query Relaxation Types

In the XML model, there are two types of query relaxations: *value relaxation* and *structure relaxation*. A value relaxation expands a value scope to allow the matching of additional answers. A structure relaxation, on the other hand, derives approximate answers by relaxing the constraint on a node or an edge in a twig. Value relaxation is orthogonal to structure relaxation. In this article, we focus on structure relaxation.

Many structure relaxation types have been proposed (8–10). We use the following three types, similar to the ones proposed in Ref. 10, which capture most of the relaxation types used in previous work.

- **Node Relabel.** With this relaxation type, a node can be relabeled to similar or equivalent labels according to domain knowledge. We use $rel(\$u, l)$ to represent a relaxation operation that renames a node $\$u$ to label l. For example, the twig in Fig. 2 can be relaxed to that in Fig. 3(a) by relabeling the node *section* to *paragraph*.
- **Edge Generalization.** With an edge relaxation, a parent-to-child edge ('/') in a twig can be generalized to an ancestor-to-descendant edge ('//'). We use $gen(e_{\$u,\$v})$ to represent a generalization of the edge between nodes $\$u$ and $\$v$. For example, the twig in Fig. 2 can be relaxed to that in Fig. 3(b) by relaxing the edge between nodes *body* and *section*.

[2]To distinguish a data node from a query node, we prefix the notation of a query node with a \$.

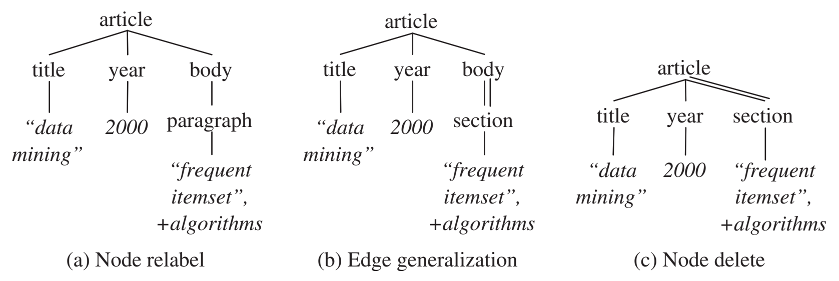


Figure 3. Examples of structure relaxations for Fig.2.

- **Node Deletion.** With this relaxation type, a node may be deleted to derive approximate answers. We use *del*(\$*v*) to denote the deletion of a node \$*v*. When \$*v* is a leaf node, it can simply be removed. When \$*v* is an internal node, the children of node \$*v* will be connected to the parent of \$*v* with ancestor-descendant edges ("//"). For instance, the twig in Fig. 2 can be relaxed to that in Fig. 3(c) by deleting the internal node *body*. As the root node in a twig is a special node representing the search context, we assume that any twig root cannot be deleted.

Given a twig T, a *relaxed twig* can be generated by applying one or more relaxation operations to T. Let m be the number of relaxation operations applicable to T, then there are at most $\binom{m}{1} + \ldots + \binom{m}{m} = 2^m$ relaxation operation combinations. Thus, there are at most 2^m relaxed twigs.

XML QUERY RELAXATION BASED ON SCHEMA CONVERSION

One approach to XML query relaxation is to convert XML schema, transform XML documents into relational tables with the converted schema, and then apply relational query relaxation techniques. A schema conversion tool, called XPRESS (*X*ml *P*rocessing and *R*elaxation in r*E*lational *S*torage *S*ystem) has been developed for these purposes: XML documents are mapped into relational formats so that queries can be processed and relaxed using existing relational technologies. Figure 4 illustrates the query relaxation flow via the schema conversion approach. This process first begins by extracting the schema information, such as DTD, from XML documents via tools such as XML Spy(see http://www.xmlspy.com.) Second, XML schema is transformed to relational schema via schema conversion ([e.g., XPRESS). Third, XML documents are parsed, mapped into tuples, and inserted into the relational databases. Then, relational query relaxation techniques [e.g., CoBase (3,6)] can be used to relax query conditions. Further, semi-structured queries over XML documents are translated into SQL queries. These SQL queries are processed and relaxed if there is no answer or there are insufficient answers available. Finally, results in the relational format are converted back into XML (e.g., the Nesting-based Translation Algorithm (NeT) and Constraints-based Translation Algorithm (CoT) (11) in XPRESS). The entire process can be done automatically and is transparent to users. In the following sections, we shall briefly describe the mapping between XML and relational schema.

Mapping XML Schema to Relational Schema

Transforming a hierarchical XML model to a flat relational model is a nontrivial task because of the following

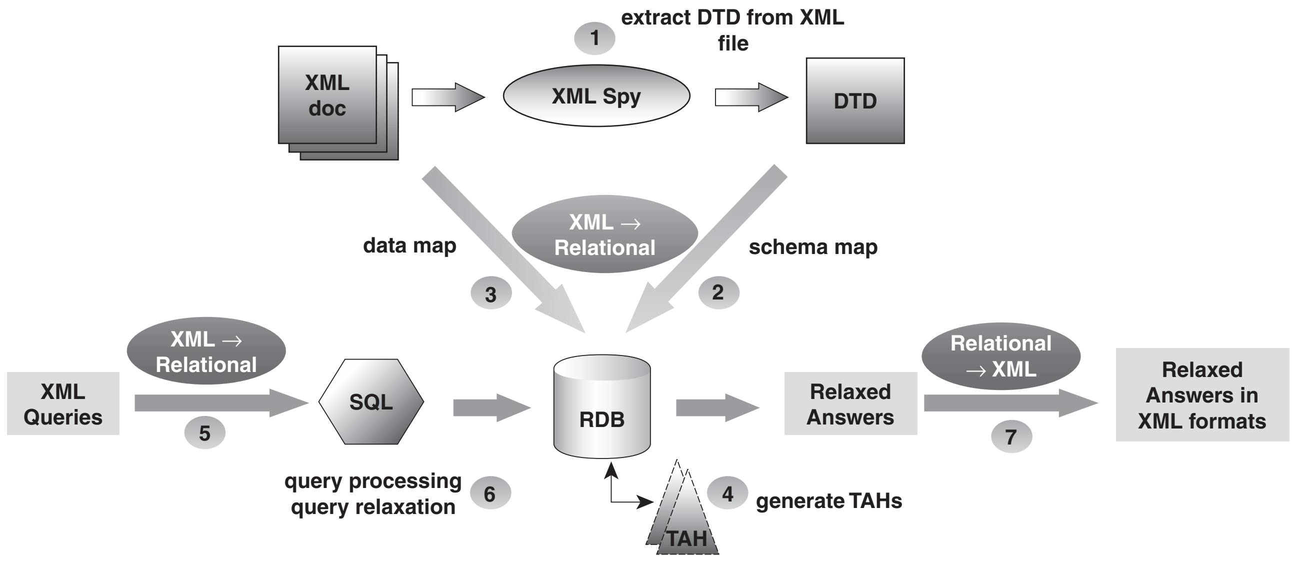


Figure 4. The processing flow of XML query relaxation via schema conversion.

inherent difficulties: the nontrivial 1-to-1 mapping, existence of set values, complicated recursion, and/or fragmentation issues. Several research works have been reported in these areas. Shanmugasundaram et al. (12) mainly focuses on the issues of structural conversion. The Constraints Preserving Inline (CPI) algorithm (13) considers the semantics existing in the original XML schema during the transformation. CPI inclines as many descendants of an element as possible into a single relation. It maps an XML element to a table when there is 1-to-$\{0,\ldots\}$ or 1-to-$\{1,\ldots\}$ cardinality between its parent and itself. The first cardinality has the semantics of "any," denoted by * in XML. The second means "at least," denoted by $+$. For example, consider the following DTD fragment:

```
<!ELEMENT author (name, address)>
<!ELEMENT name (firstname?, lastname)>
```

A naive algorithm will map every element into a separate table, leading to excessive fragmentation of the document, as follows:

```
author (address, name_id)
name (id, firstname, lastname)
```

The CPI algorithm converts the DTD fragment above into a single relational table as **author** (firstname, lastname, address).

In addition, semantics such as #REQUIRED in XML can be enforced in SQL with NOT NULL. Parent-to-child relationships are captured with KEYS in SQL to allow join operations. Figure 5 overviews the CPI algorithm, which uses a structure-based conversion algorithm (i.e., a hybrid algorithm) (13), as a basis and identifies various semantic constraints in the XML model. The CPI algorithm has been implemented in XPRESS, which reduces the number of tables generated while preserving most constraints.

Mapping Relational Schema to XML Schema

After obtaining the results in the relational format, we may need to represent them in the XML format before returning them back to users. XPRESS developed a Flat Translation (FT) algorithm (13), which translates tables in a relational schema to elements in an XML schema and columns in a relational schema to attributes in an XML schema. As FT translates the "flat" relational model to a "flat" XML model in a one-to-one manner, it does not use basic "non-flat" features provided by the XML model such as representing subelements though regular expression operator (e.g., "*" and "+"). As a result, the NeT algorithm (11) is proposed to decrease data redundancy and obtains a more intuitive schema by: (1) removing redundancies caused by multivalued dependencies; and (2) performing grouping on attributes. The NeT algorithm, however, considering tables one at a time, cannot obtain an overall picture of the relational schema where many tables are interconnected with each other through various other dependencies. The CoT algorithm (11) uses inclusion dependencies (INDs) of relational schema, such as foreign key constraints, to capture the interconnections between relational tables and represent them via parent-to-child hierarchical relationships in the XML model.

Query relaxation via schema transformation (e.g., XPRESS) has the advantage of leveraging on the well-developed relational databases and relational query relaxation techniques. Information, however, may be lost during the decomposition of hierarchical XML data into "flat" relational tables. For example, by transforming the following XML schema into the relational schema author (firstname, lastname, address), we lose the hierarchical relationship between element *author* and element *name*, as well as the information that element *firstname* is optional.

```
<!ELEMENT author (name, address)>
<!ELEMENT name (firstname?,lastname)>
```

Further, this approach does not support structure relaxations in the XML data model. To remedy these shortcomings, we shall perform query relaxation on the XML model directly, which will provide both value relaxation and structure relaxation.

A COOPERATIVE APPROACH FOR XML QUERY RELAXATION

Query relaxation is often user-specific. For a given query, different users may have different specifications about which conditions to relax and how to relax them. Most existing approaches on XML query relaxation (e.g., (10)) do not provide control during relaxation, which may yield undesired approximate answers. To provide user-specific approximate query answering, it is essential for an XML system to have a relaxation language that allows users to specify their relaxation control requirements and to have the capability to control the query relaxation process.

Furthermore, query relaxation usually returns a set of approximate answers. These answers should be ranked based on their relevancy to both the structure and the content conditions of the posed query. Most existing ranking models (e.g., (14,15)) only measure the content similarities between queries and answers, and thus are

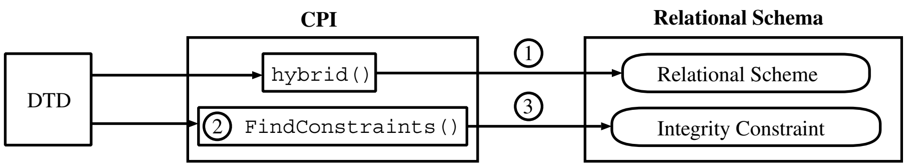


Figure 5. Overview of the CPI algorithm.

inadequate for ranking approximate answers that use structure relaxations. Recently, in Ref. (16), the authors proposed a family of structure scoring functions based on the occurrence frequencies of query structures among data without considering data semantics. Clearly, using the rich semantics provided in XML data in design scoring functions can improve ranking accuracy.

To remedy these shortcomings, we propose a new paradigm for XML approximate query answering that places users and their demands in the center of the design approach. Based on this paradigm, we develop a cooperative XML system that provides userspecific approximate query answering. More specifically, we first, develop a relaxation language that allows users to specify approximate conditions and control requirements in queries (e.g., preferred or unacceptable relaxations, nonrelaxable conditions, and relaxation orders).

Second, we introduce a relaxation index structure that clusters twigs into multilevel groups based on relaxation types and their distances. Thus, it enables the system to control the relaxation process based on users' specifications in queries.

Third, we propose a semantic-based tree editing distance to evaluate XML structure similarities, which is based on not only the number of operations but also the operation semantics. Furthermore, we combine structure and content similarities in evaluating the overall relevancy.

In Fig. 6, we present the architecture of our CoXML query answering system. The system contains two major parts: offline components for building relaxation indexes and online components for processing and relaxing queries and ranking results.

- *Building relaxation indexes.* The *Relaxation Index Builder* constructs relaxation indexes, XML Type Abstraction Hierarchy (XTAH), for a set of document collections.
- *Processing, relaxing queries, and ranking results.* When a user posts a query, the *Relaxation Engine* first sends the query to an *XML Database Engine* to search for answers that exactly match the structure conditions and approximately satisfy the content conditions in the query. If enough answers are found, the *Ranking Module* ranks the results based on their relevancy to the content conditions and returns the ranked results to the user. If there are no answers or insufficient results, then the *Relaxation Engine*, based on the user-specified relaxation constructs and controls, consults the relaxation indexes for the best relaxed query. The relaxed query is then resubmitted to the *XML Database Engine* to search for approximate answers. The *Ranking Module* ranks the returned approximate answers based on their relevancies to both structure and content conditions in the query. This process will be repeated until either there are enough approximate answers returned or the query is no longer relaxable.

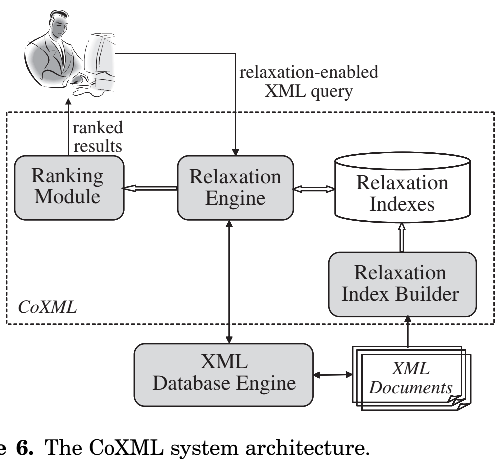


Figure 6. The CoXML system architecture.

The CoXML system can run on top of any existing XML database engine (e.g., BerkeleyDB[3], Tamino[4], DB2XML[5]) that retrieves exactly matched answers.

XML QUERY RELAXATION LANGUAGE

A number of XML approximate search languages have been proposed. Most extend standard query languages with constructs for approximate text search (e.g., XIRQL (15), TeXQuery (17), NEXI (18)). For example, TeXQuery extends XQuery with a rich set of full-text search primitives, such as proximity distances, stemming, and thesauri. NEXI introduces *about* functions for users to specify approximate content conditions. XXL (19) is a flexible XML search language with constructs for users to specify both approximate structure and content conditions. It, however, does not allow users to control the relaxation process. Users may often want to specify their preferred or rejected relaxations, nonrelaxable query conditions, or to control the relaxation orders among multiple relaxable conditions.

To remedy these shortcomings, we propose an XML relaxation language that allows users both to specify approximate conditions and to control the relaxation process. A relaxation-enabled query $\mathcal{Q}$ is a tuple ($\mathcal{T}$, $\mathcal{R}$, $\mathcal{C}$, $\mathcal{S}$), where:

- $\mathcal{T}$ is a twig as described earlier;
- $\mathcal{R}$ is a set of relaxation constructs specifying which conditions in $\mathcal{T}$ may be approximated when needed;
- $\mathcal{C}$ is a boolean combination of relaxation control stating how the query shall be relaxed; and
- $\mathcal{S}$ is a stop condition indicating when to terminate the relaxation process.

The execution semantics for a relaxation-enabled query are as follows: We first search for answers that exactly match the query; we then test the stop condition to check whether relaxation is needed. If not, we repeatedly relax

[3]See http://www.sleepycat.com/

[4]See http://www.softwareag.com/tamino

[5]See http://www.ibm.com/software/data/db2/

```
<inex_topic topic_id="267"  query_type="CAS" ct_no="113" >
<castitle>//article//fm//atl[about(., "digital libraries")]</castitle>
<description>Articles containing "digital libraries" in their title.</description>
<narrative>I'm interested in articles discussing Digital Libraries as their main subject.
Therefore I require that the title of any relevant article mentions "digital library" explicitly.
Documents that mention digital libraries only under the bibliography are not relevant, as well
as documents that do not have the phrase "digital library" in their title.</narrative>
</inex_topic>
```

Figure 8. Topic 267 in INEX 05.

the twig based on the relaxation constructs and control until either the stop condition is met or the twig cannot be further relaxed.

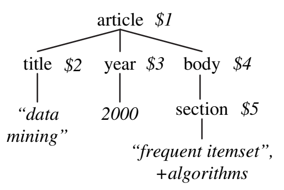
Given a relaxation-enabled query $\mathcal{Q}$, we use $\mathcal{Q}.\mathcal{T}$, $\mathcal{Q}.\mathcal{R}$, $\mathcal{Q}.\mathcal{C}$, and $\mathcal{Q}.\mathcal{S}$ to represent its twig, relaxation constructs, control, and stop condition, respectively. Note that a twig is required to specify a query, whereas relaxation constructs, control, and stop condition are optional. When only a twig is present, we iteratively relax the query based on similarity metrics until the query cannot be further relaxed.

A relaxation construct for a query $\mathcal{Q}$ is either a specific or a generic relaxation operation in any of the following forms:

- *rel*$(u,-)$, where $u \in \mathcal{Q}.\mathcal{T}.V$, specifies that node u may be relabeled when needed;
- *del*(u), where $u \in \mathcal{Q}.\mathcal{T}.V$, specifies that node u may be deleted if necessary; and
- *gen*$(e_{u,v})$, where $e_{u,v} \in \mathcal{Q}.\mathcal{T}.E$, specifies that edge $e_{u,v}$ may be generalized when needed.

The relaxation control for a query $\mathcal{Q}$ is a conjunction of any of the following forms:

- Nonrelaxable condition $!r$, where $r \in \{rel(u,-),\ del(u),\ gen\ (e_{u,v}) | u,\ v \in \mathcal{Q}.\mathcal{T}.V,\ e_{u,v} \in \mathcal{Q}, \mathcal{T}.E\}$, specifies that node u cannot be relabeled or deleted or edge $e_{u,v}$ cannot be generalized;
- $Prefer(u, l_1, \ldots, ln)$, where $u \in \mathcal{Q}.\mathcal{T}.V$ and l_i is a label $(1 \leq \mathrm{i} \leq \mathrm{n})$, specifies that node u is preferred to be relabeled to the labels in the order of $(l_1, \ldots, l_n)$;
- $Reject(u,\ l_1,\ \ldots, l_n)$, where $u \in \mathcal{Q}.\mathcal{T}.V$, specifies a set of unacceptable labels for node u;
- $RelaxOrder(r_1, \ldots, r_n)$, where $r_i \in \mathcal{Q}.\mathcal{R}.\ (1 \leq \mathrm{i} \leq \mathrm{n})$, specifies the relaxation orders for the constructs in R to be $(r_1, \ldots, r_n)$; and
- $UseRType(rt_1, \ldots, rt_k)$, where $rt_i \in \{node\_relabel,\ node\_delete,\ edge\_generalize\} (1 \leq i \leq \mathrm{k} \leq 3)$, specifies the set of relaxation types allowed to be used. By default, all three relaxation types may be used.



$R = \{gen(e_{\$4,\$5}), del(\$3)\}$

$C = !del(\$4) \wedge !gen(e_{\$1,\$2}) \wedge !gen(e_{\$1,\$4}) \wedge$ UseRType(node_delete, edge_generalize)

S = AtLeast(20)

Figure 7. A sample relaxation-enabled query.

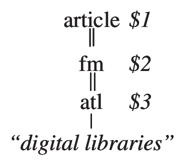
A stop condition $\mathcal{S}$ is either:

- $AtLeast(n)$, where n is a positive integer, specifies the minimum number of answers to be returned; or
- $\mathrm{d}(\mathcal{Q}.\mathcal{T}.T') \leq \tau$, where T' stands for a relaxed twig and τ a distance threshold, specifies that the relaxation should be terminated when the distance between the original twig and a relaxed twig exceeds the threshold.

Figure 7 presents a sample relaxation-enabled query. The minimum number of answers to be returned is 20. When relaxation is needed, the edge between *body* and *section* may be generalized and node *year* may be deleted. The relaxation control specifies that node *body* cannot be deleted during relaxation. For instance, a *section* about "*frequent itemset*" in an article's appendix part is irrelevant. Also, the edge between nodes *article* and *title* and the edge between nodes *article* and *body* cannot be generalized. For instance, an article with a reference to another article that possesses a title on "*data mining*" is irrelevant. Finally, only *edge generalization* and *node deletion* can be used.

We now present an example of using the relaxation language to represent query topics in INEX 05[6]. Figure 8 presents Topic 267 with three parts: *castitle* (i.e., the query formulated in an XPath-like syntax), *description*, and *narrative*. The *narrative* part describes a user's detailed information needs and is used for judging result relevancy.

The user considers an article's title (*atl*) non-relaxable and regards titles about "digital libraries" under the bibliography part (*bb*) irrelevant. Based on this narrative, we formulate this topic using the relaxation language as shown in Fig. 9. The query specifies that node *atl* cannot be relaxed (either deleted or relabeled) and node *fm* cannot be relabeled to *bb*.



$C = !rel(\$3, -) \wedge !del(\$3) \wedge Reject(\$2, bb)$

Figure 9. Relaxation specifications for Topic 267.

[6]Initiative for the evaluation of XML retrieval, See http://inex.is.informatik.uni-duisburg.de/

XML RELAXATION INDEX

Several approaches for relaxing XML or graph queries have been proposed (8,10,16,20,21). Most focus on efficient algorithms for deriving top-k approximate answers without relaxation control. For example, Amer-yahia et al. (16) proposed a DAG structure that organizes relaxed twigs based on their "consumption" relationships. Each node in a DAG represents a twig. There is an edge from twig T_A to twig T_B if the answers for T_B is a superset of those for T_A. Thus, the twig represented by an ancestor DAG node is always less relaxed and thus closer to the original twig than the twig represented by a descendant node. Therefore, the DAG structure enables efficient top-k searching when there are no relaxation specifications. When there are relaxation specifications, the approach in Ref. 16 can also be adapted to top-k searching by adding a postprocessing part that checks whether a relaxed query satisfies the specifications. Such an approach, however, may not be efficient when relaxed queries do not satisfy the relaxation specifications.

To remedy this condition, we propose an XML relaxation index structure, XTAH, that clusters relaxed twigs into multilevel groups based on relaxation types used by the twigs and distances between them. Each group consists of twigs using similar types of relaxations. Thus, XTAH enables a systematic relaxation control based on users' specifications in queries. For example, *Reject* can be implemented by pruning groups of twigs using unacceptable relaxations. *RelaxOrder* can be implemented by scheduling relaxed twigs from groups based on the specified order.

In the following, we first introduce XTAH and then present the algorithm for building an XTAH.

XML Type Abstraction Hierarchy—XTAH

Query relaxation is a process that enlarges the search scope for finding more answers. Enlarging a query scope can be accomplished by viewing the queried object at different conceptual levels.

In the relational database, a tree-like knowledge representation called Type Abstraction Hierarchy (TAH) (3) is introduced to provide systematic query relaxation guidance. A TAH is a hierarchical cluster that represents data objects at multiple levels of abstractions, where objects at higher levels are more general than objects at lower levels. For example, Fig. 10 presents a TAH for brain tumor sizes, in which a medium tumor size (i.e., 3–10 mm) is a more abstract representation than a specific tumor size

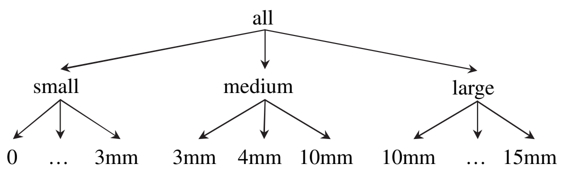


Figure 10. A TAH for brain tumor size.

(e.g., 10 mm). By such multilevel abstractions, a query can be relaxed by modifying its conditions via *generalization* (moving up the TAH) and *specialization* (moving down the TAH). In addition, relaxation can be easily controlled via TAH. For example, *REJECT* of a relaxation can be implemented by pruning the corresponding node from a TAH.

To support query relaxation in the XML model, we propose a relaxation index structure similar to TAH, called XML Type Abstraction Hierarchy (XTAH). An XTAH for a twig structure T, denoted as XT_T , is a hierarchical cluster that represents relaxed twigs of T at different levels of relaxations based on the types of operations used by the twigs and the distances between them. More specifically, an XTAH is a multilevel labeled cluster with two types of nodes: internal and leaf nodes. A leaf node is a relaxed twig of T. An internal node represents a cluster of relaxed twigs that use similar operations and are closer to each other by distance. The label of an internal node is the common relaxation operations (or types) used by the twigs in the cluster. The higher level an internal node in the XTAH, the more general the label of the node, the less relaxed the twigs in the internal node.

XTAH provides several significant advantages: (1) We can efficiently relax a query based on relaxation constructs by fetching relaxed twigs from internal nodes whose labels satisfy the constructs; (2) we can relax a query at different granularities by traversing up and down an XTAH; and (3) we can control and schedule query relaxation based on users' relaxation control requirements. For example, relaxation control such as nonrelaxable conditions, Reject or UseRType, can be implemented by pruning XTAH internal nodes corresponding to unacceptable operations or types.

Figure 11 shows an XTAH for the sample twig in Fig. 3(a).[7] For ease of reference, we associate each node in the XTAH with a unique ID, where the IDs of internal nodes are prefixed with I and the IDs of leaf nodes are prefixed with T'.

Given a relaxation operation r, let I_r be an internal node with a label $\{r\}$. That is, I_r represents a cluster of relaxed twigs whose common relaxation operation is r. As a result of the tree-like organization of clusters, each relaxed twig belongs to only one cluster, whereas the twig may use multiple relaxation operations. Thus, it may be the case that not all the relaxed twigs that use the relaxation operation r are within the group I_r. For example, the relaxed twig T'_2, which uses two operations $gen(e_{\$1,\$2})$ and $gen(e_{\$4,\$5})$, is not included in the internal node that represents $\{gen(e_{\$4,\$5})\}$, I_7, because T'_2 may belong to either group I_4 or group I_7 but is closer to the twigs in group I_4.

To support efficient searching or pruning of relaxed twigs in an XTAH that uses an operation r, we add a virtual link from internal node I_r to internal node I_k, where I_k is not a descendant of I_r, but all the twigs within I_k use operation r. By doing so, relaxed twigs that use operation r are either within group I_r or within the groups connected to I_r by virtual links. For example, internal node I_7 is connected to internal nodes I_{16} and I_{35} via virtual links.

[7]Due to space limitations, we only show part of the XTAH here.

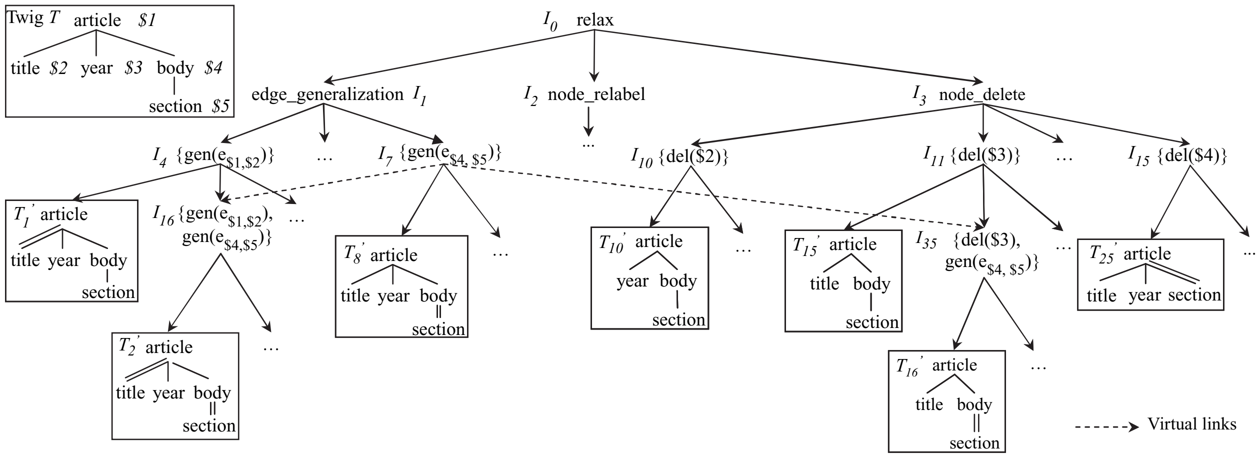

Twig T article $1
title $2 year $3 body $4
section $5
I0 relax
edge_generalization I1
I2 node_relabel
I3 node_delete
I4 {gen(e$1,$2)}
I7 {gen(e$4,$5)}
I10 {del($2)}
I11 {del($3)}
I15 {del($4)}
I16 {gen(e$1,$2), gen(e$4,$5)}
I35 {del($3), gen(e$4,$5)}
T1' article title year body section
T2' article title year body section
T8' article title year body section
T10' article year body section
T15' article title body section
T16' article title body section
T25' article title year section
Virtual links

Figure 11. An example of XML relaxation index structure for the twig T.

Thus, all the relaxed twigs using the operation $gen(e_{\$4,\$5})$ are within the groups I_7, I_{16}, and I_{35}.

Building an XTAH

With the introduction of XTAH, we now present the algorithm for building the XTAH for a given twig T.

Algorithm 1 Building the XTAH for a given twig T

```
Input: $T$: a twig
    $K$: domain knowledge about similar node labels
Output: $XT_T$ : an XTAH for $T$
 1: $RO_T \leftarrow$ GerRelaxOperations($T$, $K$) {GerRelaxOperations
    ($T$, $K$) returns a set of relaxation operations applicable to the
    twig $T$ based on the domain knowledge K}
 2: let $XT_T$ be a rooted tree with four nodes: a root node relax
    with three child nodes node_relabel, node_delete and
    edge_generalization
 3: for each relaxation operation $r \in RO_T$ do
 4:    rtype $\leftarrow$ the relaxation type of $r$
 5:    InsertXTNode(/relax/rtype, {r}) {InsertXTNode(p, n)
       inserts node $n$ into $XT_T$ under path $p$}
 6:    $T' \leftarrow$ the relaxed twig using operation $r$
 7:    InsertXTNode (/relax/rtype,/{r}, $T'$)
 8: end for
 9: for $k = 2$ to $|RO_T|$ do
10:    $S_k \leftarrow$ all possible combinations of k relaxation operations
       in $RO_T$
11:    for each combination $s \in S_k$ do
12:       let $s = \{r_1, \ldots, r_k\}$
13:       if the set of operations in $s$ is applicable to $T$ then
14:          $T' \leftarrow$ the relaxed twig using the operations in $s$
15:          $I_i \leftarrow$ the node representing $s - \{r_i\} (1 \le i \le k)$
16:          $I_j \leftarrow$ the node s.t. $\forall i, d(T', I_j) \le d(T', I_i) (1 \le i, j \le k)$
17:          InsertXTNode(///$I_j$, $\{r_1, \ldots, r_k\}$)
18:          InsertXTNode(//$I_j$/$\{r_1, \ldots, r_k\}$, $T'$)
19:          AddVLink(//$\{r_j\}$,//$I_j$) {AddV Link($p_1$,$p_2$) adds a
             virtual link from the node under path $p_1$ to the node
             under path $p_2$}
20:       end if
21:    end for
22: end for
```

In this subsection, we assume that a distance function is available that measures the structure similarity between twigs. Given any two twigs T_1 and T_2, we use $d(T_1, T_2)$ to represent the distance between the two twigs. Given a twig T and an XTAH internal node I, we measure the distance between the twig and the internal node, $d(T, I)$, as the average distance between T and any twig T′ covered by I.

Algorithm 1 presents the procedure of building the XTAH for twig T in a top-down fashion. The algorithm first generates all possible relaxations applicable to T (Line 1). Next, it initializes the XTAH with the top two level nodes (Line 2). In Lines 3–8, the algorithm generates relaxed twigs using one relaxation operation and builds indexes on these twigs based on the type of the relaxation used: For each relaxation operation r, it first adds a node to represent r, then inserts the node into the XTAH based on r's type, and places the relaxed twig using r under the node. In Lines 9–22, the algorithm generates relaxed twigs using two or more relaxations and builds indexes on these twigs. Let s be a set of k relaxation operations ($k \ge 2$), T′ a relaxed twig using the operations in s, and I an internal node representing s. Adding node I into the XTAH is a three-step process: (1) it first determines I's parent in the XTAH (Line 16). In principle, any internal node that uses a subset of the operations in s can be I's parent. The algorithm selects an internal node I_j to be I's parent if the distance between T' and I_j is less than the distance between T' and other parent node candidates; (2) It then connects node I to its parent I_j and adds a leaf node representing T' to node I (Lines 17 and 18). (3) Finally, it adds a virtual link from the internal node representing the relaxation operation r_j to node I (Line 19), where r_j is the operation that occurs in the label of I but not in label of its parent node I_j .

QUERY RELAXATION PROCESS

Query Relaxation Algorithm

Algorithm 2 Query Relaxation Process

```
Input: $XT_T$: an XTAH
    $\mathcal{Q} = \{\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{S}\}$: a relaxation-enabled query
Output: $\mathcal{A}$: a list of answers for the query $\mathcal{Q}$
1: $\mathcal{A} \leftarrow$ SearchAnswer($\mathcal{Q}.T$); {Searching for exactly matched answers for $\mathcal{Q}.T$}
2: if (the stop condition $\mathcal{Q}.\mathcal{S}$ is met) then
3:     return $\mathcal{A}$
4: end if
5: if (the relaxation controls $\mathcal{Q}.\mathcal{C}$ are non-empty) then
6:     PruneXTAH($XT_T$, $\mathcal{Q}.\mathcal{C}$) {Pruning nodes in $XT_T$ that contain relaxed twigs using unacceptable relaxation operations based on $\mathcal{Q}.\mathcal{C}$}
7: end if
8: if the relaxation constructs $\mathcal{Q}.\mathcal{R}$ are non-empty then
9:     while ($\mathcal{Q}.\mathcal{S}$ is not met)&&(not all the constructs in $\mathcal{Q}.\mathcal{R}$ have been processed) do
10:        $T' \leftarrow$ the relaxed twig from $XT_T$ that best satisfies the relaxation specifications $\mathcal{Q}.\mathcal{R}$ & $\mathcal{Q}.\mathcal{C}$
11:        Insert SearchAnswer($T'$) into $\mathcal{A}$
12:    end while
13: end if
14: while ($\mathcal{Q}.\mathcal{T}$ is relaxable)&&($\mathcal{Q}.\mathcal{S}$ is not met) do
15:    $T' \leftarrow$ the relaxed twig from $XT_T$ that is closest to $\mathcal{Q}.\mathcal{T}$ based on distance
16:    Insert SearchAnswer($T'$) into $\mathcal{A}$
17: end while
18: return $\mathcal{A}$
```

Figure 12 presents the control flow of a relaxation process based on XTAH and relaxation specifications in a query. The *Relaxation Control* module prunes irrelevant XTAH groups corresponding to unacceptable relaxation operations or types and schedules relaxation operations based on *Prefer* and *RelaxOrder* as specified in the query. Algorithm 2 presents the detailed steps of the relaxation process:

1. Given a relaxation-enabled query $\mathcal{Q} = \{\mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{S}\}$ and an XTAH for $\mathcal{Q}.\mathcal{T}$, the algorithm first searches for exactly matched answers. If there are enough number of answers available, there is no need for relaxation and the answers are returned (Lines 1–4).
2. If relaxation is needed, based on the relaxation control $\mathcal{Q}.\mathcal{C}$ (Lines 5–7), the algorithm prunes XTAH internal nodes that correspond to unacceptable operations such as nonrelaxable twig nodes (or edges), unacceptable node relabels, and rejected relaxation types. This step can be efficiently carried out by using internal node labels and virtual links. For example, the relaxation control in the sample query (Figure 7) specifies that only *node_delete* and *edge_generalization* may be used. Thus, any XTAH node that uses *node_relabel*, either within group I_2 or connected to I_2 by virtual links, is disqualified from searching. Similarly, the internal nodes I_{15} and I_4, representing the operations $del(\$4)$ and $gen(e_{\$1,\ \$2})$, respectively, are pruned from the XTAH by the *Relaxation Control* module.
3. After pruning disqualified internal groups, based on relaxation constructs and control, such as *RelaxOrder* and *Prefer*, the *Relaxation Control* module schedules and searches for the relaxed query that best satisfies users' specifications from the XTAH. This step terminates when either the stop condition is met or all the constructs have been processed. For example, the sample query contains two relaxation constructs: $gen(e_{\$4,\$5})$ and $del(\$3)$. Thus, this step selects the best relaxed query from internal groups, I_7 and I_{11}, representing the two constructs, respectively.
4. If further relaxation is needed, the algorithm then iteratively searches for the relaxed query that is closest to the original query by distance, which may use relaxation operations in addition to those specified in the query. This process terminates when either the stop condition holds or the query cannot be further relaxed.
5. Finally, the algorithm outputs approximate answers.

Searching for Relaxed Queries in an XTAH

We shall now discuss how to efficiently search for the best relaxed twig that has the least distance to the query twig from its XTAH in Algorithm 2.

A brute-force approach is to select the best twig by checking all the relaxed twigs at the leaf level. For a twig T with m relaxation operations, the number of relaxed twigs can be up to 2^m. Thus, the worst case time complexity for this approach is $O(2^m)$, which is expensive.

To remedy this condition, we propose to assign representatives to internal nodes, where a representative summarizes the distance characteristics of all the relaxed twigs covered by a node. The representatives facilitate the searching for the best relaxed twig by traversing an

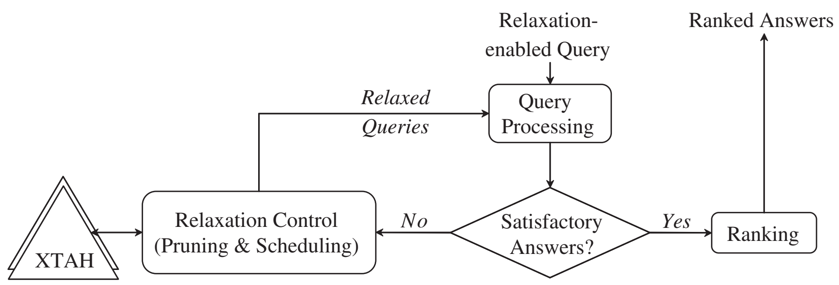


Figure 12. Query relaxation control flow.

XTAH in a top-down fashion, where the path is determined by the distance properties of the representatives. By doing so, the worst case time complexity of finding the best relaxed query is $O(d * h)$, where d is the maximal degree of an XTAH node and h is the height of the XTAH. Given an XTAH for a twig T with m relaxation operations, the maximal degree of any XTAH node and the depth of the XTAH are both $O(m)$. Thus, the time complexity of the approach is $O(m^2)$, which is far more efficient than the brute-force approach ($O(2^m)$).

In this article, we use M-tree (22) for assigning representatives to XTAH internal nodes. M-tree provides an efficient access method for similarity search in the "metric space," where object similarities are defined by a distance function. Given a tree organization of data objects where all the data objects are at the leaf level, M-tree assigns a data object covered by an internal node I to be the representative object of I. Each representative object stores the covering radius of the internal node (i.e., the maximal distance between the representative object and any data object covered by the internal node). These covering radii are then used in determining the path to a data object at the leaf level that is closest to a query object during similarity searches.

XML RANKING

Query relaxation usually generates a set of approximate answers, which need to be ranked before being returned to users. A query contains both structure and content conditions. Thus, we shall rank an approximate answer based on its relevancy to both the structure and content conditions of the posed query. In this section, we first present how to compute XML content similarity, then describe how to measure XML structure relevancy, and finally discuss how to combine structure relevancy with content similarity to produce the overall XML ranking.

XML Content Similarity

Given an answer $\mathcal{A}$ and a query $\mathcal{Q}$, the content similarity between the answer and the query, denoted as cont_sim(A and $\mathcal{Q}$), is the sum of the content similarities between the data nodes and their corresponding matched query nodes. That is,

$$cont\_sim(\mathcal{A}, \mathcal{Q}) = \sum_{v \in \mathcal{A}, \$u \in \mathcal{Q}.\mathcal{T}.v, u \; matches \; \$u} cont\_sim(v, \$u) \tag{1}$$

For example, given the sample twig in Fig. 2, the set of nodes {1, 2, 6, 7, 8} in the sample data tree is an answer. The content similarity between the answer and the twig equals to $cont\_sim(2, \$2) + cont\_sim(6, \$3) + cont\_sim(8, \$5)$.

We now present how to evaluate the content similarity between a data node and a query node. Ranking models in traditional IR evaluate the content similarity between a document to a query and thus need to be extended to evaluating the content similarity between an XML data node and a query node. Therefore, we proposed an extended vector space model (14) for measuring XML content similarity, which is based on two concepts: *weighted term frequency* and *inverse element frequency*.

Weighted Term Frequency. Due to the hierarchical structure of the XML data model, the text of a node is also considered as a part of the ancestor nodes' text, which introduces the challenge of how to calculate the content relevancy of an XML data node v to a query term t, where t could occur in the text of any node nested within the node v. For example, all three *section* nodes (i.e., nodes 8, 10, and 12) in the XML data tree (Fig. 1) contain the phrase "frequent itemsets" in their text parts. The phrase "frequent itemsets" occurs at the *title* part of the node 8, the *paragraph* part in the node 10, and the *reference* part in the node 12. The same term occurring at the different text parts of a node may be of different weights. For example, a "frequent itemset" in the *title* part of a *section* node has a higher weight than a "frequent itemset" in the *paragraph* part of a *section* node, which, in turn, is more important than a "frequent itemset" in the *reference* part of a *section* node. As a result, it may be inaccurate to measure the weight of a term t in the text of a data node v by simply counting the occurrence frequency of the term t in the text of the node v without distinguishing the term's occurrence paths within the node v.

To remedy this condition, we introduce the concept of "weighted term frequency," which assigns the weight of a term t in a data node v based on the term's occurrence frequency and the weight of the *occurrence path*. Given a data node v and a term t, let $p = v_1.v_2 \ldots .v_k$ be an *occurrence path* for the term t in the node v, where v_k is a descendant node of v, v_k directly contains the term t, and $v \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$ represents the path from the node v to the node v_k. Let $w(p)$ and $w(v_i)$ denote the weight for the path p and the node v_i, respectively. Intuitively, the weight of the path $p = v_1.v_2 \ldots .v_k$ is a function of the weights of the nodes on the path (i.e., $w(p) = \mathrm{f}(w(v_1), \ldots w(v_k)))$, with the following two properties:

1. $\mathrm{f}(w(v_1), w(v_2), \ldots, w(v_k))$ is a monotonically increasing function with respect to $w(v_i)$ $(1 \leq i \leq k)$; and
2. $\mathrm{f}(w(v_1), w(v_2), \ldots, w(v_k))) = 0$ if any $w(v_i) = 0$ $(1 \leq i \leq k)$.

The first property states that the path weight function is a monotonically increasing function. That is, the weight of a path is increasing if the weight of any node on the path is increasing. The second property states that if the weight of any node on the path is zero, then the weight of the path is zero. For any node v_i $(1 \leq i \leq k)$ on the path p, if the weight of the node v_i is zero, then it implies that users are not interested in the terms occurring under the node v_i. Therefore, any term in the text of either the node v_i or a descendant node of v_i is irrelevant.

A simple implementation of the path weight function $\mathrm{f}(w(v_1), w(v_2), \ldots, w(v_k))$ that satisfies the properties stated above is to let the weight of a path equal to the

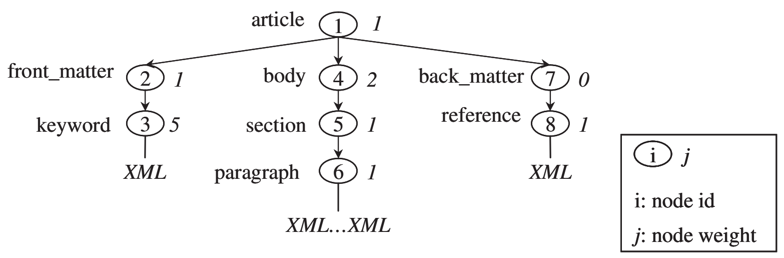


Figure 13. An example of weighted term frequency.

product of the weights of all nodes on the path:

$$w(p) = \prod_{i=1}^{k} w(v_i) \qquad (2)$$

With the introduction of the weight of a path, we shall now define the weighted term frequency for a term t in a data node v, denoted as $tf_w(v, t)$, as follows:

$$tf_w(v,t) = \sum_{j=1}^{m} w(p_j) \times tf(v, p_j, t) \qquad (3)$$

where m is the number of paths in the data node v containing the term t and $tf\,(v, p_j, t)$ is the frequency of the term t occurred in the node v via the path p_j.

For example, Fig. 13 illustrates an example of an XML data tree with the weight for each node shown in italic beside the node. The weight for the *keyword* node is 5 (i.e., $w(keyword) = 5$). From Equation (2), we have $w(front\_matter.keyword) = 5{*}1 = 5$, $w(body.section.paragraph) = 2{*}1{*}1 = 2$, and $w(back\_matter.reference) = 0{*}1 = 0$, respectively. The frequencies of the term "XML" in the paths *front_matter.keyword*, *body.section.paragraph*, and *back_ matter.reference* are 1, 2, and 1, respectively. Therefore, from Equation (3), the weighted term frequency for the term "XML" in the data node *article* is $5{*}1 + 2{*}2 + 0{*}1 = 9$.

Inverse Element Frequency. Terms with different popularity in XML data have different degrees of discriminative power. It is well known that a term frequency (*tf*) needs to be adjusted by the inverse document frequency (*idf*) (23). A very popular term (with a small *idf*) is less discriminative than a rare term (with a large *idf*). Therefore, the second component in our content ranking model is the concept of "inverse element frequencys" (*ief*), which distinguishes terms with different discriminative powers in XML data. Given a query Q and a term t, let $\$u$ be the node in the twig $Q.\mathcal{T}$ whose content condition contains the term t (i.e., $t \in \$u.cont$). Let *DN* be the set of data nodes such that each node in *DN* matches the structure condition related with the query node $\$u$. Intuitively, the more frequent the term t occurs in the text of the data nodes in *DN*, the less discriminative power the term t has. Thus, the inverse element frequency for the query term t can be measured as follows:

$$ief(\$u,\, t) = log\left(\frac{N_1}{N_2} + 1\right) \qquad (4)$$

where N_1 denotes the number of nodes in the set *DN* and N_2 represents the number of the nodes in the set *DN* that contain the term t in their text parts.

For example, given the sample XML data tree (Fig. 1) and the query twig (Fig. 2), the inverse element frequency for the term "frequent itemset" can be calculated as follows: First, the content condition of the query node \$5 contains the term "frequent itemset"; second, there are three data nodes (i.e., nodes 8, 10, and 12) that match the query node \$5; and third, all the three nodes contain the term in their text. Therefore, the inverse element frequency for the term "frequent itemset" is log(3/3 + 1) = log2. Similarly, as only two nodes (i.e., nodes 8 and 12) contain the term "algorithms," the inverse element frequency for the term "algorithms" is log(3/2 + 1) = log(5/2).

Extended Vector Space Model. With the introduction "weighted term frequency" and "inverse element frequency," we now first present how we compute the content similarity between a data node and a query node and then present how we calculate the content similarity between an answer and a query.

Given a query node $\$u$ and a data node v, where the node v matches the structure condition related with the query node $\$u$, the content similarity between the nodes v and $\$u$ can be measured as follows:

$$cont\_sim(v, \$u) = \sum_{t \in \$u.cont} w(m(t)) \times tf_w(v,t) \times ief(\$u, t) \qquad (5)$$

where t is a term in the content condition of the node $\$u$, $m(t)$ stands for the modifier prefixed with the term t (e.g., "+", " ", "−"), and $w(m(t))$ is the weight for the term modifier as specified by users.

For example, given the *section* node, \$5, in the sample twig (Fig. 2), the data node 8 in Fig. 1 is a match for the twig node \$5. Suppose that the weight for a "+" term modifier is 2 and the weight for the *title* node is 5, respectively. The content similarity between the data node 8 and the twig node \$5 equals to tf_w(8, *"frequent itemset"*)× ief(\$5, *"frequent itemset"*) + w('+') × tf_w(8, *"algorithms"*) × *ief*(\$5, *"algorithms"*), which is 5 × log2 + 2 × 5 × log(5/2) = 18.22. Similarly, the data node 2 is a match for the twig node *title* (i.e., \$2) and the content similarity between them is tf_w(2, *"data mining"*) × ief(\$2, "data mining") = 1.

Discussions. The extended vector space model has shown to be very effective in ranking content similarities of SCAS retrieval results[8](14). SCAS retrieval results are usually of relatively similar sizes. For example, for the twig in Fig. 2, suppose that the node *section* is the target node (i.e., whose matches are to be returned as answers). All the SCAS retrieval results for the twig will be sections inside article bodies. Results that approximately match the twig, however, could be nodes other than *section* nodes, such as *paragraph*, *body*, or *article* nodes, which are of varying sizes. Thus, to apply the extended vector space model for evaluating content similarities of approximate answers under this condition, we introduce the factor of "weighted sizes" into the model for normalizing the biased effects caused by the varying sizes in the approximate answers (24):

$$cont\_sim(\mathcal{A}, \mathcal{Q}) = \sum_{v \in \mathcal{A}, \$u \in \mathcal{Q}.\mathcal{T}.\mathcal{V}, v\, matches\, \$u} \frac{cont\_sim(v, \$u)}{log_2\, wsize(v)} \tag{6}$$

where *wsize*(v) denotes the weighted size of a data node v.

Given an XML data node v, *wsize*(v) is the sum of the number of terms directly contained in node v's text, *size*-($v.text$), and the weighted size of all its child nodes adjusted by their corresponding weights, as shown in the following equation.

$$wsize(v) = size(v.text) + \sum_{v_i\ s.t.\ v|v_i} wsize(v_i) * w(v_i) \tag{7}$$

For example, the weighted size of the *paragraph* node equals the number of terms in its text part, because the *paragraph* node does not have any child node.

Our normalization approach is similar to the scoring formula proposed in Ref. 25, which uses the log of a document size to adjust the product of tf and *idf*.

Semantic-based Structure Distance

The structure similarity between two twigs can be measured using tree editing distance (e.g., (26)), which is frequently used for evaluating tree-to-tree similarities. Thus, we measure the structure distance between an answer $\mathcal{A}$ and a query $\mathcal{Q}$, *struct_dist*($\mathcal{A}$, $\mathcal{Q}$), as the editing distance between the twig $\mathcal{Q}\cdot\mathcal{T}$ and the least relaxed twig T', $d(\mathcal{Q}.\mathcal{T}, T')$, which is the total costs of operations that relax $\mathcal{Q}.\mathcal{T}$ to T':

$$struct\_dist(\mathcal{A}, \mathcal{Q}) = d(\mathcal{Q}.\mathcal{T}, T') = \sum_{i=1}^{k} cost(r_i) \tag{8}$$

where $\{r_1, \ldots, r_k\}$ is the set of operations that relaxes $\mathcal{Q}.\mathcal{T}$ to T' and $cost(r_i)$ $(0 \le cost(r_i) \le 1)$ is equal to the cost of the relaxation operation $r_i (1 \le \mathrm{i} \le \mathrm{k})$.

Existing edit distance algorithms do not consider operation cost. Assigning equal cost to each operation is simple, but does not distinguish the semantics of different operations. To remedy this condition, we propose a semantic-based relaxation operation cost model.

We shall first present how we model the semantics of XML nodes. Given an XML dataset D, we represent each data node v_i as a vector $\{w_{i1}, w_{i2}, \ldots, w_{iN}\}$, where N is the total number of distinct terms in D and w_{ij} is the weight of the jth term in the text of v_i. The weight of a term may be computed using *tf*idf*(23) by considering each node as a "document." With this representation, the similarity between two nodes can be computed by the cosine of their corresponding vectors. The greater the cosine of the two vectors, the semantically closer the two nodes.

We now present how to model the cost of an operation based on the semantics of the nodes affected by the operation with regard to a twig T as follows:

- Node Relabel – *rel*(u, l)
  A node relabel operation, *rel*(u, l), changes the label of a node u from $u.label$ to a new label l. The more semantically similar the two labels are, the less the relabel operation will cost. The similarity between two labels, $u.label$ and l, denoted as *sim*($u.label$, l), can be measured as the cosine of their corresponding vector representations in XML data. Thus, the cost of a relabel operation is:

  $$cost(rel(u, l)) = 1 - sim(u.lable,\ l) \tag{9}$$

  For example, using the INEX 05 data, the cosine of the vector representing *section* nodes and the vector representing *paragraph* nodes is 0.99, whereas the cosine of the vector for *section* nodes and the vector for *figure* nodes is 0.38. Thus, it is more expensive to relabel node *section* to *paragraph* than to *figure*.
- Node Deletion – *del*(u)
  Deleting a node u from the twig approximates u to its parent node in the twig, say v. The more semantically similar node u is to its parent node v, the less the deletion will cost. Let $V_{v/u}$ and V_v be the two vectors representing the data nodes satisfying v/u and v, respectively. The similarity between v/u and v, denoted as *sim*(v/u, v), can be measured as the cosine of the two vectors $V_{v/u}$ and V_v. Thus, a node deletion cost is:

  $$cost(del(u)) = 1 - sim(v/u,\ u) \tag{10}$$

  For example, using the INEX 05 data, the cosine of the vector for *section* nodes inside *body* nodes and the vector for *body* nodes is 0.99, whereas the cosine of the vector for *keyword* nodes inside *article* nodes and the vector for *article* nodes is 0.2714. Thus, deleting the *keyword* node in Fig. 3(a) costs more than deleting the *section* node.
- Edge Generalization – *gen*($e_{v,u}$)
  Generalizing the edge between nodes $\$v$ and $\$u$ approximates a child node v/u to a descendant node $v//u$. The closer v/u is to $v//u$ in semantics, the less the edge generalization will cost. Let $V_{v/u}$ and $V_{v//u}$ be two vectors representing the data nodes satisfying

[8]In a SCAS retrieval task, structure conditions must be matched exactly whereas content conditions are to be approximately matched.

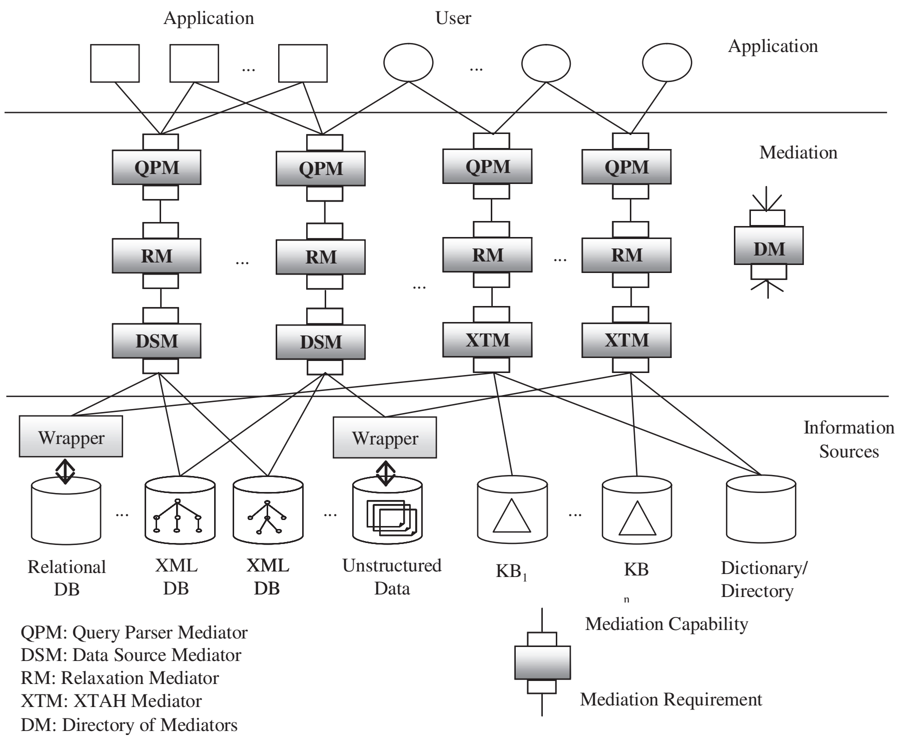


Figure 14. A scalable and extensible cooperative XML query answering system.

v/u and $v//u$, respectively. The similarity between v/u and $v//u$, denoted as $sim(v/u, v//u)$, can be measured as the cosine of the two vectors $V_{v/u}$ and $V_{v//u}$. Thus, the cost for an edge generalization can be measured as:

$$cost(gen(e_{v,u})) = 1 - sim(v/u, v//u) \qquad (11)$$

For example, relaxing *article/title* in Fig. 3(a) to *article//title* makes the title of an article's author (i.e., */article/author/title*) an approximate match. As the similarity between an article's title and an author's title is low, the cost of generalizing *article/title* to *article//title* may be high.

Note that our cost model differs from Amer-Yahia et al. (16) in that Amer-Yahia et al. (16) applies *idf* to twig structures without considering node semantics, whereas we applied *tf*idf* to nodes with regard to their corresponding data content.

The Overall Relevancy Ranking Model

We now discuss how to combine structure distance and content similarity for evaluating the overall relevancy.

Given a query $\mathcal{Q}$, the relevancy of an answer $\mathcal{A}$ to the query $\mathcal{Q}$, denoted as $sim(\mathcal{A}, \mathcal{Q})$, is a function of two factors: the structure distance between $\mathcal{A}$ and $\mathcal{Q}$ (i.e., $struct\_dist(\mathcal{A}, \mathcal{Q})$), and the content similarity between $\mathcal{A}$ and $\mathcal{Q}$, denoted as $cont\_sim(\mathcal{A}, \mathcal{Q})$. We use our extended vector space model for measuring content similarity (14). Intuitively, the larger the structure distance, the less the relevancy; the larger the content similarity, the greater the relevancy. When the structure distance is zero (i.e., exact structure match), the relevancy of the answer to the query should be determined by their content similarity only. Thus, we combine the two factors in a way similar to the one used in XRank (27) for combining element rank with distance:

$$sim(\mathcal{A}, \mathcal{Q}) = \alpha^{struct\_dist(\mathcal{A},\mathcal{Q})} * cost\_sim(\mathcal{A}, \mathcal{Q}) \qquad (12)$$

where α is a constant between 0 and 1.

A SCALABLE AND EXTENSIBLE ARCHITECTURE

Figure 14 illustrates a mediator architecture framework for a cooperative XML system. The architecture consists of an application layer, a mediation layer, and an information source layer. The information source layer includes a set of heterogeneous data sources (e.g., relational databases, XML databases, and unstructured data), knowledge bases, and knowledge base dictionaries or directories. The knowledge base dictionary (or directory) stores the characteristics of all the knowledge bases, including XTAH and domain knowledge in the system. Non-XML data can be converted into the XML format by wrappers. The mediation layer consists of data source mediators, query parser mediators, relaxation mediators, XTAH mediators, and directory mediators. These mediators are selectively interconnected to meet the specific application requirements. When the demand for certain mediators increases, additional copies of the mediators can be added to reduce the loading. The mediator architecture allows incremental growth with application, and thus the system is *scalable*. Further,

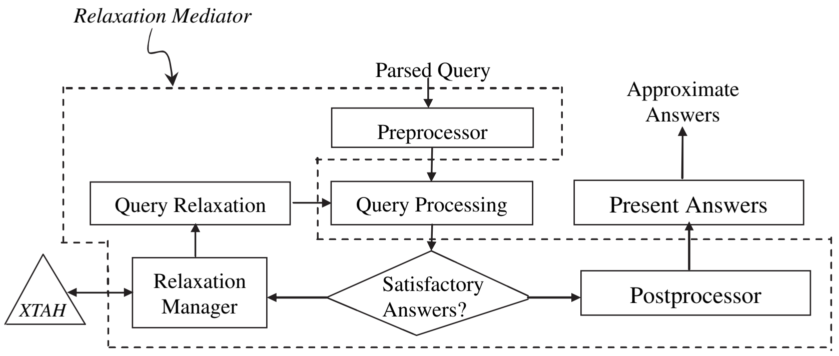


Figure 16. The flow chart of XML query relaxation processing.

different types of mediators can be interconnected and can communicate with each other via a common communication protocol (e.g., KQML (28), FIPA[9]) to perform a joint task. Thus, the architecture is *extensible*.

For query relaxation, based on the set of frequently used query tree structures, the XTAHs for each query tree structure can be generated accordingly. During the query relaxation process, the XTAH manager selects the appropriate XTAH for relaxation. If there is no XTAH available, the system generates the corresponding XTAH on-the-fly.

We shall now describe the functionalities of various mediators as follows:

- **Data Source Mediator (DSM)**
  The data source mediator provides a virtual database interface to query different data sources that usually have different schema. The data source mediator maintains the characteristics of the underlying data sources and provides a unified description of these data sources. As a result, XML data can be accessed from data sources without knowing the differences of the underlying data sources.
- **Query Parser Mediator (PM)**
  The query parser mediator parses the queries from the application layer and transforms the queries into query representation objects.
- **Relaxation Mediator (RM)**
  Figure 15 illustrates the functional components of the relaxation mediator, which consists of a pre-processor, a relaxation manager, and a post-processor. The flow of the relaxation process is depicted in Fig. 16. When a relaxation-enabled query is presented to the relaxation mediator, the system first goes through a pre-processing phase. During pre-processing, the system transforms the relaxation constructs into standard XML query constructs. All relaxation control operations specified in the query are processed and forwarded to the relaxation manager and are ready for use if the query requires relaxation. The modified query is then presented to the underlying databases for execution. If no answers are returned, then the relaxation manager relaxes the query conditions guided by the relaxation index (XTAH). We repeat the relaxation process until either the stop condition is met or the query is no longer relaxable. Finally, the returned answers are forwarded to the post-processing module for ranking.
- **XTAH Mediator (XTM)**
  The XTAH mediator provides three conceptually separate, yet interlinked functions to peer mediators: XTAH Directory, the XTAH Management, and the XTAH Editing facilities, as illustrated in Fig. 17.

  Usually, a system contains a large number of XTAHs. To allow other mediators to determine which XTAHs exist within the system and their characteristics, the XTAH mediator contains a directory. This directory is searchable by the XML query tree structures.

  The XTAH management facility provides client mediators with traversal functions and data extraction functions (for reading the information out of XTAH nodes). These capabilities present a common interface so that peer mediators can traverse and extract data from an XTAH. Further, the XTAH

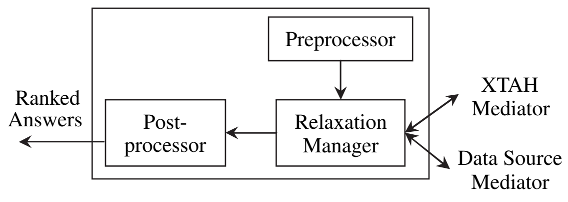


Figure 15. The relaxation mediator.

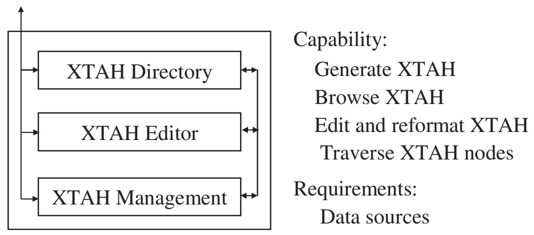


Figure 17. The XTAH mediator.

[9]See http://www.fipa.org.

mediator has an editor that allows users to edit XTAHs to suit their specific needs. The editor handles recalculation of all information contained within XTAH nodes during the editing process and supports exportation and importation of entire XTAHs if a peer mediator wishes to modify it.

- **Directory Mediator (DM)**
  The directory mediator provides the locations, characteristics, and functionalities of all the mediators in the system and is used by peer mediators for locating a mediator to perform a specific function.

A COOPERATIVE XML (CoXML) QUERY ANSWERING TESTBED

A CoXML query answering testbed has been developed at UCLA to evaluate the effectiveness of XML query relaxation through XTAH. Figure 18 illustrates the architecture of CoXML testbed, which consists of a query parser, a preprocessor, a relaxation manager, a database manager, an XTAH manager, an XTAH builder, and a post-processor. We describe the functionality provided by each module as follows:

- *XTAH Builder*. Given a set of XML documents and the domain knowledge, the *XTAH builder* constructs a set of XTAHs that summarizes the structure characteristics of the data.
- *Query Parser*. The *query parser* checks the syntax of the query. If the syntax is correct, then it extracts information from the parsed query and creates a query representation object.
- *Preprocessor*. The pre-processor transforms relaxation constructs (if any) in the query into the standard XML query constructs.

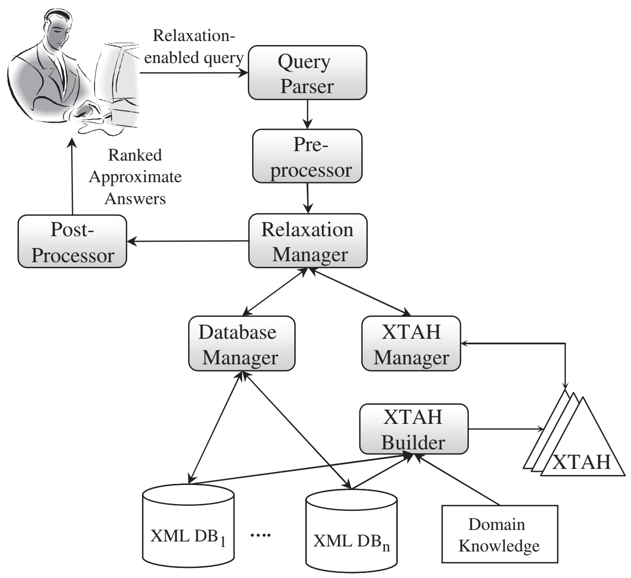


Figure 18. The architecture of the CoXML testbed.

- *Relaxation Manager*. The relaxation manager performs the following services: (1) building a relaxation structure based on the specified relaxation constructs and controls; (2) obtaining the relaxed query conditions from the XTAH Manager; (3) modifying the query accordingly; and (4) retrieving the exactly matched answers.
- *Database Manager*. The database manager interacts with an XML database engine and returns exactly matched answers for a standard XML query.
- *XTAH Manager*. Based on the structure of the query tree, the XTAH manager selects an appropriate XTAH to guide the query relaxation process.
- *Post-processor*. The post-processor takes unsorted answers as input, ranks them based on both structure and content similarities, and outputs a ranked list of results.

EVALUATION OF XML QUERY RELAXATION

INEX is a DELOS working group[10] that aims to provide a means for evaluating XML retrieval systems in the form of a large heterogeneous XML test collection and appropriate scoring methods. The INEX test collection is a large set of scientific articles, represented in XML format, from publications of the IEEE Computer Society covering a range of computer science topics. The collection, approximately 500 megabytes, contains over 12,000 articles from 18 magazines/transactions from the period of 1995 to 2004, where an article (on average) consists of 1500 XML nodes. Different magazines/transactions have different data organizations, although they use the same ontology for representing similar content.

There are three types of queries in the INEX query sets: content-only (CO), strict content and structure (SCAS), and vague content and structure (VCAS). CO queries are traditional information retrieval (IR) queries that are written in natural language and constrain the content of the desired results. Content and structure queries not only restrict content of interest but also contain either explicit or implicit references to the XML structure. The difference between a SCAS and a VCAS query is that the structure conditions in a SCAS query must be interpreted exactly whereas the structure conditions in a VCAS query may be interpreted loosely.

To evaluate the relaxation quality of the CoXML system, we perform the VCAS retrieval runs on the CoXML testbed and compare the results against the INEX's relevance assessments for the VCAS task, which can be viewed as the "gold standard." The evaluaion studies reveal the expressiveness of the relaxation language and the effectiveness of using XTAH in providing user-desired relaxation control. The evaluation results demonstrate that our content similarity model has significantly high precision at low recall regions. The model achieves the highest average precision as compared with all the 38 official submissions in

[10]See http://www.iei.pi.cnr.it/DELOS

INEX 03 (14). Furthermore, the evaluation results also demonstrate that using the semantic-based distance function yields results with greater relevancy than using the uniform-cost distance function. Comparing with other systems in INEX 05, our user-centeric relaxation approach retrieves approximate answers with greater relevancy (29).

SUMMARY

Approximate matching of query conditions plays an important role in XML query answering. There are two approaches to XML query relaxation: either through schema conversion or directly through the XML model. Converting the XML model to the relational model by schema conversion can leverage on the mature relational model techniques, but information may be lost during such conversions. Furthermore, this approach does not support XML structure relaxation. Relaxation via the XML model approach remedies these shortcomings. In this article, a new paradigm for XML approximate query answering is proposed that places users and their demands at the center of the design approach. Based on this paradigm, we develop an XML system that cooperates with users to provide user-specific approximate query answering. More specifically, a relaxation language is introduced that allows users to specify approximate conditions and relaxation control requirements in a posed query. We also develop a relaxation index structure, XTAH, that clusters relaxed twigs into multilevel groups based on relaxation types and their interdistances. XTAH enables the system to provide user-desired relaxation control as specified in the query. Furthermore, a ranking model is introduced that combines both content and structure similarities in evaluating the overall relevancy of approximate answers returned from query relaxation. Finally, a mediatorbased CoXML architecture is presented. The evaluation results using the INEX test collection reveal the effectiveness of our proposed user-centric XML relaxation methodology for providing user-specific relaxation.

ACKNOWLEDGMENTS

The research and development of CoXML has been a team effort. We would like to acknowledge our CoXML members, Tony Lee, Eric Sung, Anna Putnam, Christian Cardenas, Joseph Chen, and Ruzan Shahinian, for their contributions in implementation, testing, and performance evaluation.

BIBLIOGRAPHY

1. S. Boag, D. Chamberlin, M. F Fernandez, D. Florescu, J. Robie, and J. S. (eds.), XQuery 1.0: An XML Query Language. Available http://www.w3.org/TR/xquery/.
2. W. W Chu ,Q. Chen, and A. Huang, Query Answering via Cooperative Data Inference. *J. Intelligent Information Systems (JIIS)*, **3** (1): 57–87, 1994.
3. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson, CoBase: A scalable and extensible cooperative information system. *J. Intell. Inform. Syst.*, **6** (11), 1996.
4. S. Chaudhuri and L. Gravano, Evaluating Top-k Selection Queries. In *Proceedings of 25th International Conference on Very Large Data Bases*, September 7–10, 1999, Edinburgh, Scotland, UK.
5. T. Gaasterland, Cooperative answering through controlled query relaxation, *IEEE Expert*, **12** (5): 48–59, 1997.
6. W.W. Chu, Cooperative Information Systems, in B. Wah (ed.), *The Encyclopedia of Computer Science and Engineering*, New York: Wiley, 2007.
7. Y. Kanza, W. Nutt, and Y. Sagiv, Queries with Incomplete Answers Over Semistructured Data. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 31 – June 2, 1999, Philadelphia, Pennsylvania.
8. Y. Kanza and Y. Sagiv, In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 21–23, 2001, Santa Barbara, California.
9. T. Schlieder, In *Proceedings of 10th International Conference on Extending Database Technology*, March 26–31, 2006, Munich, Germany.
10. S. Amer-Yahia, S. Cho, and D. Srivastava, XML Tree Pattern Relaxation. In *Proceedings of 10th International Conference on Extending Database Technology*, March 26–31, 2006, Munich, Germany.
11. D. Lee, M. Mani, and W. W Chu, Schema Conversions Methods between XML and Relational Models, *Knowledge Transformation for the Semantic Web. Frontiers in Artificial Intelligence and Applications* Vol. 95, IOS Press, 2003, pp. 1–17.
12. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases*, September 7–10, 1999, Edinburgh, Scotland, UK.
13. D. Lee and W.W Chu, CPI: Constraints-preserving Inlining algorithm for mapping XML DTD to relational schema, *J. Data and Knowledge Engineering, Special Issue on Conceptual Modeling*, **39** (1): 3–25, 2001.
14. S. Liu, Q. Zou, and W. Chu, Configurable Indexing and Ranking for XML Information Retrieval. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, July 25–29, 2004, Sheffield, UK.
15. N. Fuhr and K. Groβjohann, XIRQL: A Query Language for Information Retrieval in XML Documents. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval,* September 9–13, 2001, New Orleans Louisiana.
16. S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman, Structure and Content Scoring for XML. In *Proceedings of the 31st International Conference on Very Large Data Bases*, August 30– September 2, 2005, Trondheim, Norway.
17. S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, TeXQuery: A Full-Text Search Extension to XQuery. In *Proceedings of 13th International World Wide Web Conference.* May 17–22, 2004, New York.
18. A. Trotman and B. Sigurbjornsson, Narrowed Extended XPath I NEXI. In *Proceedings of the 3rd Initiative of the Evaluation of XML Retrieval (INEX 2004) Workshop*, December 6–8, 2004, Schloss Dagstuhl, Germany,
19. A. Theobald and G. Weikum, Adding Relevance to XML. In *Proceedings of the 3rd International Workshop on the Web and*

Databases, WebDB 2000, Adam's, May 18–19, 2000, Dallas, Texas.

20. A. Marian, S. Amer-Yahia, N. Koudas, and D. Srivastava, Adaptive Processing of Top-k Queries in XML. *In Proceedings of the 21st International Conference on Data Engineering, ICDE 2005*, April 5–8, 2005, Tokyo, Japan.
21. I. Manolescu, D. Florescu, and D. Kossmann, Answering XML Queries on Heterogeneous Data Sources. *In Proceedings of 27th International Conference on Very Large Data Bases*, September 11–14, 2001, Rome, Italy.
22. P. Ciaccia, M. Patella, and P. Zezula, M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. *In Proceedings of 23rd International Conference on Very Large Data Bases*, August 25–29, 1997, Athens, Greece.
23. G. Salton and M. J McGill, *Introduction to Modern Information Retrieval*, New York: McGraw-Hill, 1983.
24. S. Liu, W. Chu, and R. Shahinian, Vague Content and Structure Retrieval(VCAS) for Document-Centric XML Retrieval. *Proceedings of the 8th International Workshop on the Web and Databases (WebDB 2005)*, June 16–17, 2005, Baltimore, Maryland.
25. W. B Frakes and R. Baeza-Yates, *Information Retreival: Data Structures and Algorithms*, Englewood Cliffs, N.J.: Prentice Hall, 1992.
26. K. Zhang and D. Shasha, Simple fast algorithms for the editing distance between trees and related problems, *SIAM J. Comput.*, **18** (6):1245– 1262, 1989.
27. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, XRANK: Ranked Keyword Search Over XML Document. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, June 9–12, 2003, San Diego, California.
28. T. Finin, D. McKay, R. Fritzson, and R. McEntire, KQML: An information and knowledge exchange protocol, in K. Fuchi and T. Yokoi, (eds), *Knowledge Building and Knowledge Sharing*, Ohmsha and IOS Press, 1994.
29. S. Liu and W. W Chu, CoXML: A Cooperative XML Query Answering System. Technical Report # 060014, Computer Science Department, UCLA, 2006.
30. T. Schlieder and H. Meuss, Querying and ranking XML documents, *J. Amer. So. Inf. Sci. Technol.*, **53** (6):489.
31. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton, Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

WESLEY W. CHU
SHAORONG LIU
University of California,
Los Angeles
Los Angeles, California