# The Relaxation Manager for XML Query Relaxation

**Anna Putnam**
**302-771-439**

**UCLA Computer Science Department**
**Masters Comprehensive Exam**

**Wednesday, June 16, 2004**

# Table of Contents

## Abstract

XML is emerging as the de facto standard for storing structured documents on the Web. With the new wealth of information stored in XML documents comes the need to query these documents. The complexity and heterogeneity of XML data sources, however makes effective query answering difficult. Often the user has strong semantic knowledge about what they are looking for, but has difficulties in knowing the specifics of how the data is structured. XML query relaxation relieves this stress on the user by relaxing constructs in a query allowing more answers to be returned. In this paper we will explore XML relaxation techniques as well as CoXML (cooperative XML) [1], a query processing engine utilizing the power of query relaxation. In particular, we will focus on the Relaxation Manager Module of CoXML. A preliminary version of the Relaxation Manager has been implemented in Java. This module is responsible for building a relaxation specification for a relaxation query as well as controlling the relaxation process.

## 1 Introduction

XML (eXtensible Markup Language) is a structured document language becoming the standard for the sharing of data on the web. With this increase in use must also come ways to efficiently handle XML data. Query Languages, in particular must be able to effectively and efficiently query an XML document in order to return interesting results to users of the document. Query Relaxation is a way to expand traditional query methods to allow flexibility in query answering for XML documents.

The use of XML as a data source has properties that make query relaxation particularly important. The schema of an XML model can be very large. The hieratical structure of XML documents makes the number and complexity of schemas possible virtually infinite. Moreover, users of such a system cannot be required to know the entire schema. Users are far more likely to have a high level understanding of the data represented in the XML documents than they are to understand the nuances of the hieratical structure used.

A query relaxation scheme must then allow users to submit queries that can be relaxed to fit the structure of the data.

Another property of XML documents that makes query relaxation essential is the number of heterogeneous XML data sources available. The same type of information can be structured in many different ways that users cannot be expected to know. Instead users should provide what they are looking for without specifying how it is stored.

Previous work on XML query relaxation has focused on first transforming the XML data to relational tables. Queries are then answered and relaxed on the relational system. This approach has the major drawback of losing semantic information about the XML document in the conversion. By answering queries on the XML itself, the query answering system can relax both the *values* and the *structure* of the nodes in the XML document.

In this paper, we examine query relaxation techniques for XML data. We begin by briefly explaining the XML data model including XML query languages. We then move on to discuss various relaxation types for XML documents including both value relaxations and structure relaxations. We also discuss Relaxation Control operators used to specify how an XML query can be relaxed.

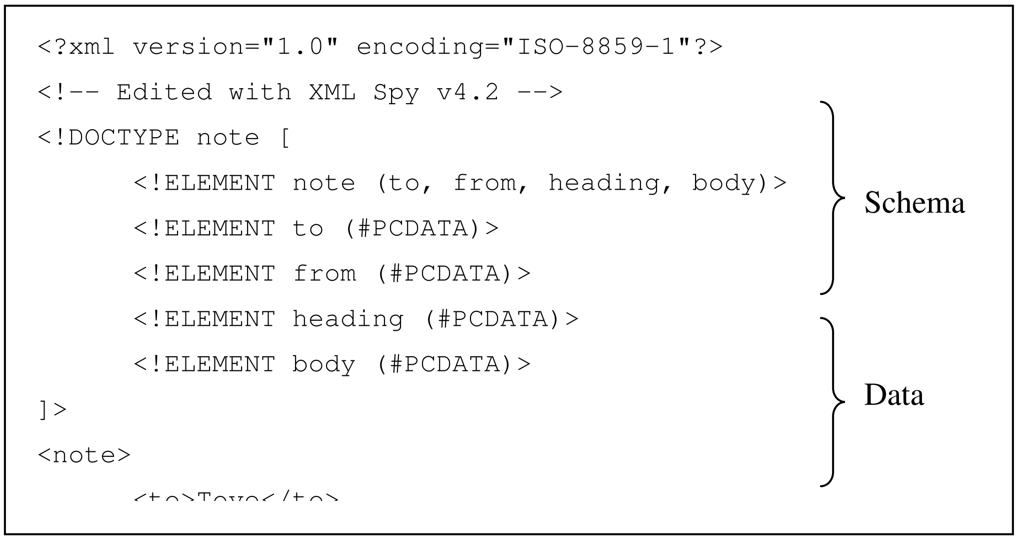
The main focus of the paper is CoXML [1], a XML query engine that incorporates these relaxation techniques. Specifically, we will describe the techniques used by the Relaxation Manager to relax XML queries. This module is the core of the query relaxation process providing methods to build relaxation strategies that will guide the order and type of relaxation.

# 2 XML

XML (eXtensible Markup Language) is a format for specifying structured documents and data. XML is called extensible because it allows users to define their own schema unlike HTML which is a pre-defined markup language. With XML one can define their own customized markup language for different types of documents. XML has become a popular way to display and distribute structured data on the web because of its flexibility. In this section we will introduce the XML data model as well as XML query languages for getting information stored in XML.

## 2.1 XML Data Model

XML is a hierarchical data model consisting of two parts: the *schema*, and the *data*. In XML, the *schema* describes the structure of the *data*. For example, in Figure 1, the XML for a simple note is shown. Here we see the schema defined first with elements such as to, from and body. We then see the data, stored hieratically surrounded by starting and ending tags defined by the schema.



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Edited with XML Spy v4.2 -->
<!DOCTYPE note [
      <!ELEMENT note (to, from, heading, body)>
      <!ELEMENT to (#PCDATA)>
      <!ELEMENT from (#PCDATA)>
      <!ELEMENT heading (#PCDATA)>
      <!ELEMENT body (#PCDATA)>
]>
<note>
      <to>Tove</to>
```

Schema
Data

*Figure 1: Example XML Document*

A popular language used for XML schemas definitions is DTD (Document Type Definition). In DTD, elements and attributes defined by the keywords <!ELEMENT> and <!ATTRIBUTE> respectively. Elements are the main building blocks of XML documents. Once Elements are defined for a given XML document, elements can be marked up by tags. Attributes provide extra information about elements and are placed inside element tags and come in name/value pairs. The following is the general BNF syntax for Element and Attribute components:

```
<!ELEMENT> <elem-name> <elem-content-model>
<!ATTLIST> <attr-name> <attr-type> <attr-option>
```

The Element content model is used to define the general structure of the element. Regular expressions can be used to describe the cardinality of parts of an element such as:

- "?" (0 or 1 instance)
- "*" (0 or many instances)
- "+" (1 or many instances).

For example, the following XML Element describes a paper with one title sub-element, one or more author sub-element, and zero or more citation sub-elements:

```
<!ELEMENT paper (title, author+, citation*)>
```

Once the schema has been defined, the remainder of the XML document contains data. Each data element has a starting and ending tag defined by the schema. For example the above Element schema may have the following data:

```
<paper>
          <title> XML Query Relaxation </title>
          <author> Anna Putnam </author>
          <citation> CoXML </citation>
</paper>
<paper>
          <title> CoXML </title>
          <author> Shaorong Liu </author>
          <author> Wesely W. Chu </author>
</paper>
```

## 2.2 XML Query Languages

Several XML Query languages have been proposed in an attempt to keep up with the growing popularity of XML. Here we will not concentrate on one particular language but rather refer to queries from a high level.

An XML document can be represented as an ordered tree with nodes representing elements and attributes, and edges representing inclusion relationships. In the XML tree, nodes are given unique ids to identify specific data elements.

Similarly, an XML query can be represented as a tree with edges of two types. "/" is used to represent parent-child relationships while "//" is used for ancestor-descendent relationships.

For example, the XML in Figure 2a can be represented by the tree shown in Figure 2b. A possible query is shown in Figure 3c. Here the query wishes to find an element "a" with a descendent "b" and a child "c". An answer to this query based on the XML tree shown in Figure 2b would be the triple of ids (1,3,4).

```
<a>
     <d>
          <b/>
     </d>
```

*Figure 2a: Sample XML*

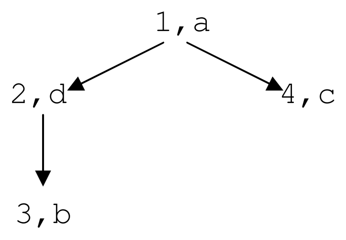


*Figure 2b: Sample XML Tree*

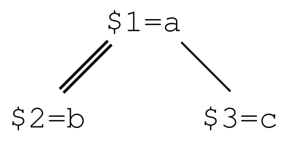


*Figure 2c: Sample XML Query*

# 3 Relaxation Types

XML query relaxation can be categorized into two main types; *value* relaxation and *structure* relaxation. In *value* relaxation, values are relaxed to expand the scope values are allowed to take. In *structure* relaxation, the structure of the query tree can be relaxed. By relaxing the *values* and/or *structure* of an XML query the system can weaken the constraints of a query and possibly return more answers to the user.

## 3.1 Value Relaxation

In value relaxation, the scope of a value is expanded to allow additional answers to be returned by a query. The value of a constraint can be relaxed to a range of numeric values or a set of non-numeric values.

For example, the query in Figure 3a can be relaxed to the query in Figure 3b. Here we see the year range is expanded from 2001-2003 to 1998-2003. The relaxed query is less restrictive than the original query and therefore can allow more answers to be returned.

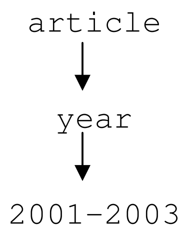


***Figure 3a: Value Query***

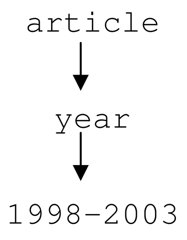


***Figure 3b: Relaxed Query***

Value relaxations for non-numeric values relax a value in a query to its next level abstraction. For example, the query in Figure 4a can be relaxed to the query in Figure 4b. Here “XML” is relaxed to “Semi-structured Data”. Again the relaxed query is less restrictive than the original query allowing more answers to be returned.

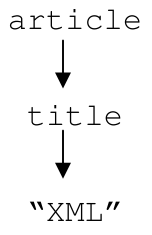


***Figure 4a: Value Query***

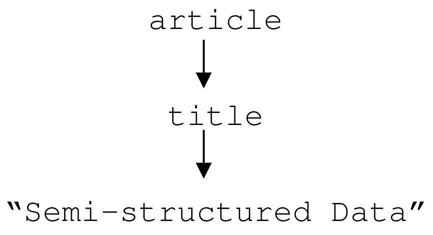


***Figure 4b: Relaxed Query***

## *3.2 Structure Relaxation*

In structure relaxation nodes and/or edges of the query tree can be relaxed to allow for more answers. There are three types of structural relaxation; *edge* relaxation, *node* relaxation, and *order* relaxation. Each type changes the query tree in some way weakening the constraints of the query posed.

### 3.2.1 Edge Relaxation

Edge relaxation relaxes an edge of the query tree to make the query less restrictive. Here, a parent-child ("/") edge can be relaxed to an ancestor-descendent edge ("//"). The set of answers returned for an ancestor-descendent relationship is a superset of the set of answers returned for a parent-child relationship.

For example, Figure 5 shows an example XML tree. Suppose the user poses the query "a/b/c". That is, an element "a" with a child "b" whose child is "c". With this query, the triple (1,7,8) is returned. If we now apply edge relaxation, we see we have three possible query relaxation results: "a//b/c", "a/b//c" and the most relaxed "a//b//c". The results of each relaxed query are shown in Figure 6. Note that each relaxed query returns a set of answers that include the original exact match answer of (1,7,8).

$$\begin{tikzcd}
 & & 1,a \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] & & \\
2,b \arrow[d] & 4,b \arrow[d] & 7,b \arrow[d] & 9,d \arrow[d] & 12,d \arrow[d] \\
3,d & 5,d \arrow[d] & 8,c & 10,b \arrow[d] & 13,b \arrow[d] \\
 & 6,c & & 11,c & 14,d \arrow[d] \\
 & & & & 15,c
\end{tikzcd}$$

***Figure 5: XML Tree for Edge Relaxation***

Original query:

“a/b/c” → (1,7,8)

Relaxed queries:

“a//b/c” → (1,7,8) & (1,10,11)

“a/b//c” → (1,7,8) & (1,4,6)

“a//b//c” → (1,7,8), (1,10,11), (1,4,6), (1,13,15)

***Figure 6: Edge Relaxation Results***

### 3.2.2 Node Relaxation

Node relaxation involves replacing a node in the query tree with a more relaxed node. Nodes in the query tree can be relaxed in several ways.

A node can be relabeled with a similar tag name based on the domain knowledge. Here, a thesaurus can be used to cluster similar terms. For example, suppose we had the domain knowledge about section-like tags as shown in Figure 7. Now if we issue the query “article/sec”, we can relax it to “article/section” therefore allowing for more answers.

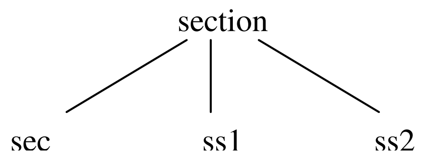


***Figure 7: Equivalent names for section-like tags***

A second way to relax a node is to replace it with a "don't care" such that it will match any non-null answer. For example, the query "/a/b/c" can be relaxed to "a/_ /c". In this example, the original query will only match one path, namely "a/b/c" whereas the relaxed query could return paths like "a/d/c", "a/e/c" etc.

Finally, a node can be relaxed by removing it while still ensuring the "superset" property. When a node *n* is a leaf node, the node can simple be removed. For example, the query "a/b/c" can be relaxed to "a/b". Here we see that the answers returned by the relaxed query is a superset of the answers returned by the original query.

If, on the other hand, node *n* is an internal node, the children of *n* will be connected to the parent of *n* by an ancestor-descendent relationship ("//"). For example, consider the query shown in Figure 8a. This query can be relaxed to the query shown in Figure 8b by removing the node "header" and connecting "article" and "title" with an ancestor-descendent edge.

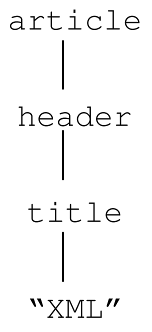


***Figure 8a: Original Query***

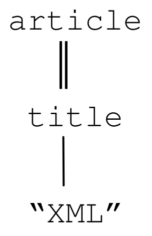


***Figure 8b: Relaxed Query***

### 3.2.3 Order Relaxation

The order in an XML query can also be relaxed to allow more answers to be returned. In an ordered XML model, the ordering of siblings is important. To weaken this constraint, we can relax ordered queries to have no ordering restrictions.

For example, the query show in Figure 9a is looking for an element "a" with descendent "b" followed by a child "c". In the figure "<" is used to indicate ordering. To relax this query, we remove the ordering constraint so that the order that element "b" and element "c" occur is unimportant. This query is shown in figure 9b.

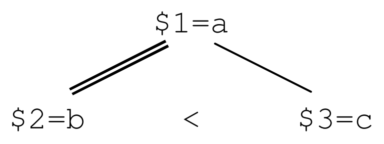


*Figure 9a: Original Ordered Query*

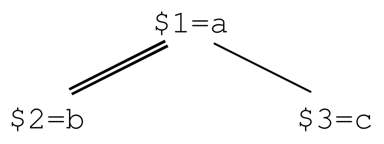
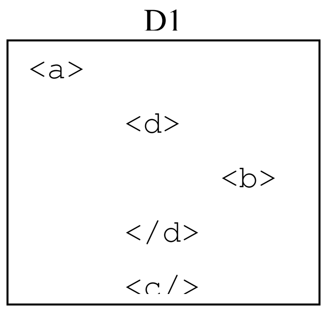


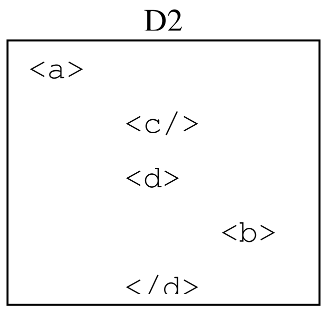
*Figure 9b: Relaxed (un-ordered) Query*

If we now examine the two XML documents shown in Figure 10, we see that the original query matches D1 only while the relaxed query matches both D1 and D2.



D1

```
<a>
    <d>
        <b>
    </d>
    <c/>
```



D2

```
<a>
    <c/>
    <d>
        <b>
    </d>
```

*Figure 10: Sample Documents D1 & D2*

## 3.3 Relaxation Control

XML Query Relaxation enables the query answering system to relax certain query conditions and consequently return more answers to the user. In addition, an XML query relaxation system can provide some relaxation control operators to give users control over how their query will be relaxed. Users can specify non-relaxable nodes and/or structures, the order in which to relax conditions, and the rank that should be applied to relaxed queries.

### 3.3.1 Non-Relaxable

Non-relaxable conditions can be denoted by "!". This operator is used to specify conditions that should not be relaxed. Such non-relaxable conditions could be nodes or structures. For example, the following query finds *book* elements with a sub-element *title* with value "XML" and sub-element *year* with value larger than 2000. In this query the *year* constraint cannot be relaxed.

```
for $b in //book
where $b/title = "XML" and $b/year > !2000
return {$b};
```

The non-relaxable operator can also be used in applied to information scenarios. For example, the following integrates *courses* with a sub-element *teacher* and sub-elements *title* and *TA*, where the edge between *course* and *title*, and *TA* are non-relaxable.

```
INTEGRATE
for $c in //course
where $c/teacher and $c!/title and $c!/TA
```

### 3.3.2 RELAX-ORDER

Using the RELAX-ORDER, a user can control the order in which conditions will be relaxed. That is, for a list of conditions that are relaxable, the user can specify the order to relax them. For example, the following query finds books that are either 1) published by "Addison-Wesley" in 2000 (condition t1) or 2) written by "John Smith" (condition t2). If there are not enough exact matches, t1 will be relaxed and then t2 will be relaxed.

```
for $b in //book
where ($b/publisher = "Addison-Wesley" and
          $b/year = 2000) CONDITION t1
     or ($b/author = "John Smith") CONDITION t2
return {$b}
RELAX-ORDER[t1,t2]
```

The RELAX-ORDER operator can also be used to control the order approximation types are applied in information integration. For example, the following query integrates XML fragments about *course* elements with sub-elements *title*, *teacher* and *TA*. The order of the structural relaxation types is 1) node re-label 2) edge approximation 3) node deletion and 4) hierarchical order approximation.

```
INTEGRATE
for $c in //course
where $c/title and $c/teacher and $/TA
RELAX-ORDER(node re-label, edge approximation, node
     deletion, hierarchical order)
```

### 3.3.3 RANK-BY

The RANK-BY operator is used to control the ranking of results. This is done by specifying a nearness metric to be used in comparing how "good" answers are in relation to the query posed.

# 4 CoXML [1]

CoXML (Cooperative XML Query Answering) is a system for answering XML queries. CoXML performs two types of functions: document indexing and query processing. Figure 11 shows the overall framework for CoXML.

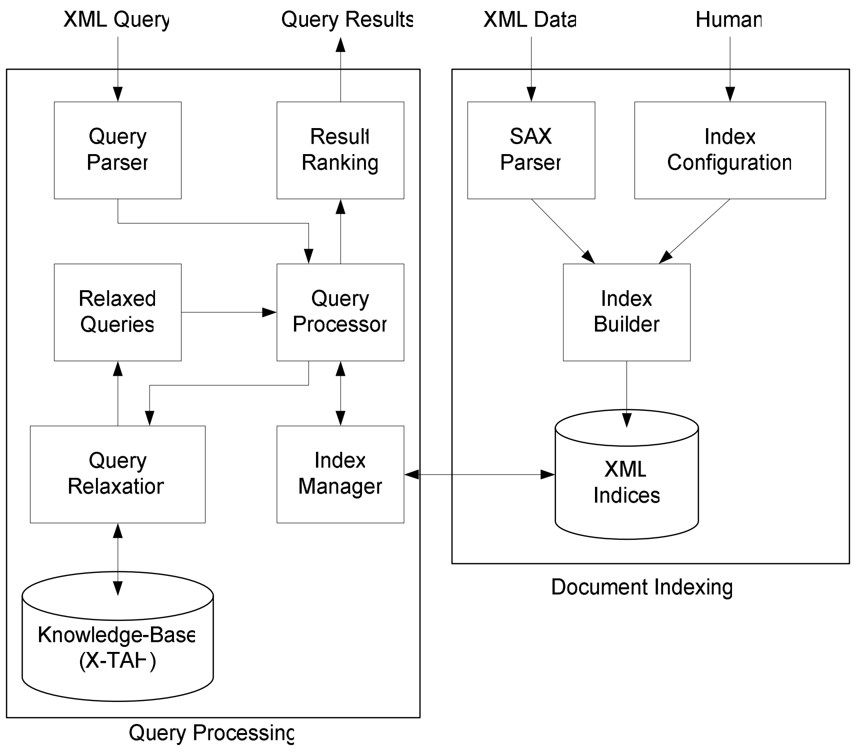


***Figure 11: The CoXML System Architecture***

In this paper we will focus on the query processing piece.

## *4.1 Knowledge Base*

The knowledge base is an important part of the CoXML system. It enables query relaxation and consists of two parts; the domain ontology and the knowledge-based XML relaxation index.

### 4.1.1 Domain Ontology

In section 3.2.2 we discussed node relaxation. To aid in node re-labeling, the relaxation engine must have a guide for which tag names are related. In CoXML, domain ontology provides the semantic relationships among tag names.

For example, Figure 12 gives the domain ontology for the equivalent names for paragraph-like tags.

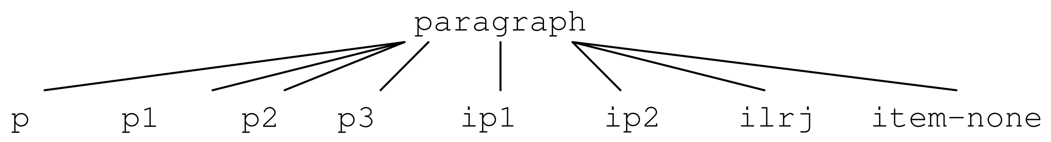


*Figure 12: Equivalent names for paragraph-like tags*

### 4.1.2 Knowledge-based XML Relaxation Index (X-TAH)

To enable the value and structure relaxations discussed in Section 3, CoXML utilizes two types of relaxation index structures, called XML Type Abstract Hierarchies (X-TAHs). One index is used for value relaxations while the other is used for structure relaxations.

An X-TAH is a tree-like abstraction hierarchy that represents the value and structure characteristics of an XML data tree. An X-TAH has internal nodes and leaf nodes. Internal nodes are abstractions of the objects in that cluster. Leaf nodes are objects in a cluster. Figure 13 shows an example X-TAH.

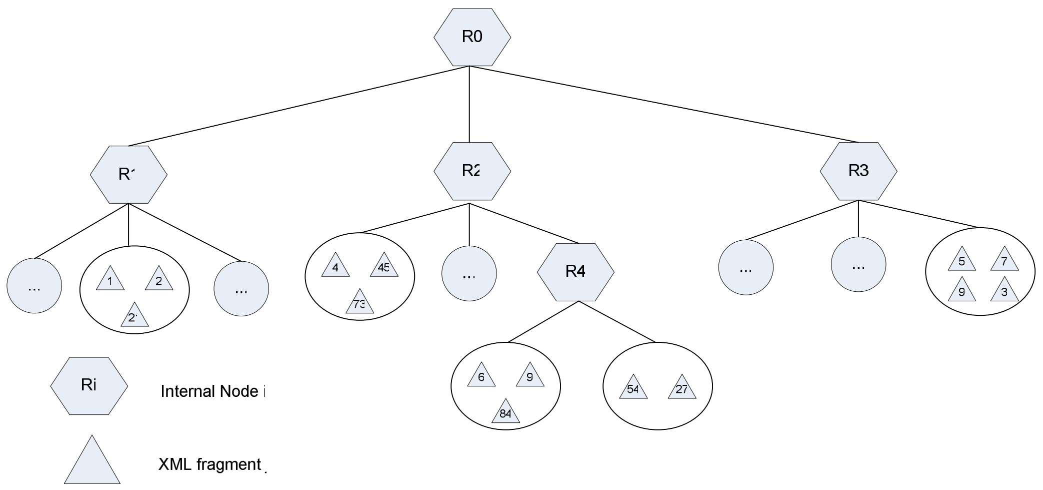


*Figure 13: Example X-TAH*

For value X-TAHs, leaf nodes are values and for structure X-TAHs, leaf nodes are structure fragments. Figure 14 shows an example X-TAH. For example, the X-TAH in Figure 14 shows the value type abstraction hierarchy for figure numbers.

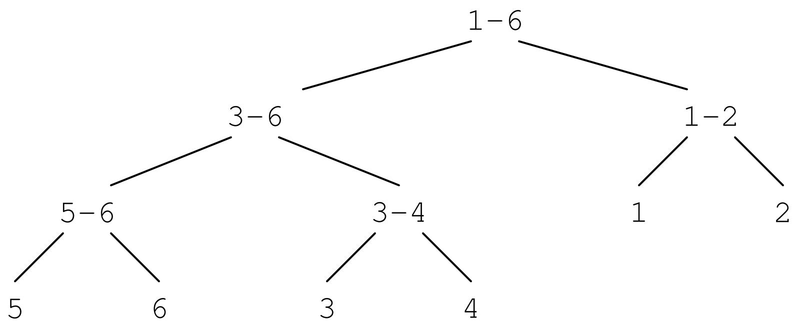
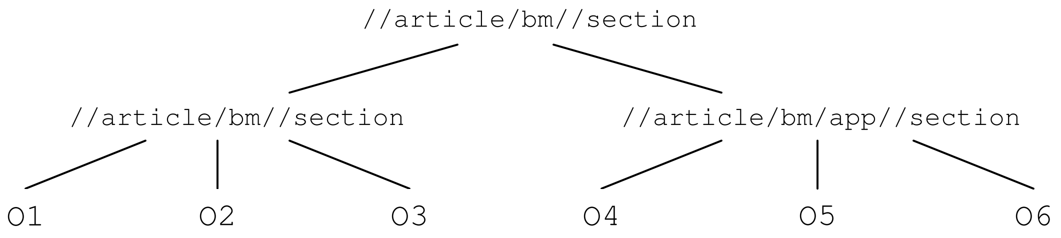


***Figure 14: Value X-TAH for Figure numbers***

As another example, the X-TAH in Figure 15 shows the structure type abstraction hierarchy for articles.



```
O1: //article/bm/sec              O4: //article/bm/app/sec
O2: //article/bm/sec/ss1          O5: //article/bm/app/sec/ss1
```

***Figure 15: Structure X-TAH for Articles***

An X-TAH can be generated by clustering similar XML fragments in XML data trees. Here “similar” is determined based on the inter-XML fragment distance. A set of XML fragments belongs to the same XML fragment pattern if any of them can be transformed into the others using relaxation operations. Once similar XML fragments have been identified, they can be clustered by either an agglomerative or divisive clustering algorithm.

## 4.2 Query Processing and Relaxation

The control flow for query processing and relaxation is shown in Figure 16. A query topic is first translated a query tree the processor can follow. The *Query Processor* then follows the data tree and executes the query, returning a set of results. If the processor returns enough results, the results are ranked and returned to the user. If, on the other hand there are not enough results, the *Query Relaxation Manager* relaxes the query based on information given by the *Knowledge Base* (X-TAH). The relaxed query is then resubmitted to the *Query Processor* to be executed. This cycle continues until enough results are given or the query can not be further relaxed.

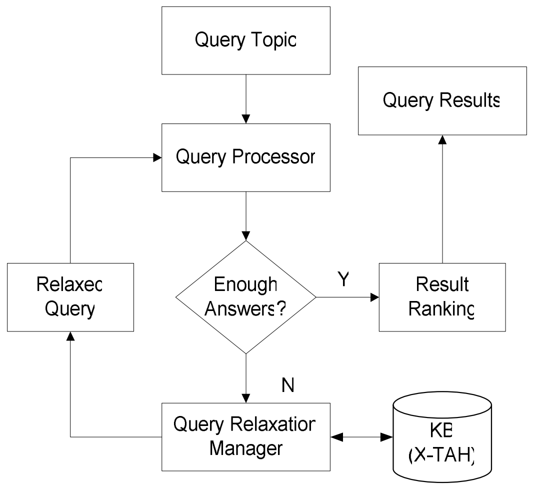


***Figure 16: The control flow of CoXML query processing***

### 4.2.1 RLXQuery Relaxation

CoXML uses the RLXQuery framework for answering XML queries. Central to this framework is the transformation of the XML query into its object representation (XQueryRep). The XQueryRep object is a hierarchy of objects with each sub-object representing a piece of the RlxQuery. This representation allows parts of the query to be examined and updated in the relaxation process.

For example, consider the simple query:

```
for $b in document("books.xml")//book
where $b/title = "XML" OR $b/year > 2002
return $b
```

This query will be represented by the XQueryRep object show in *Figure 17.* In the figure the top object represents the FOR/LET clause of the Query, the middle object represents the WHERE clause, and the third object represents the return value.

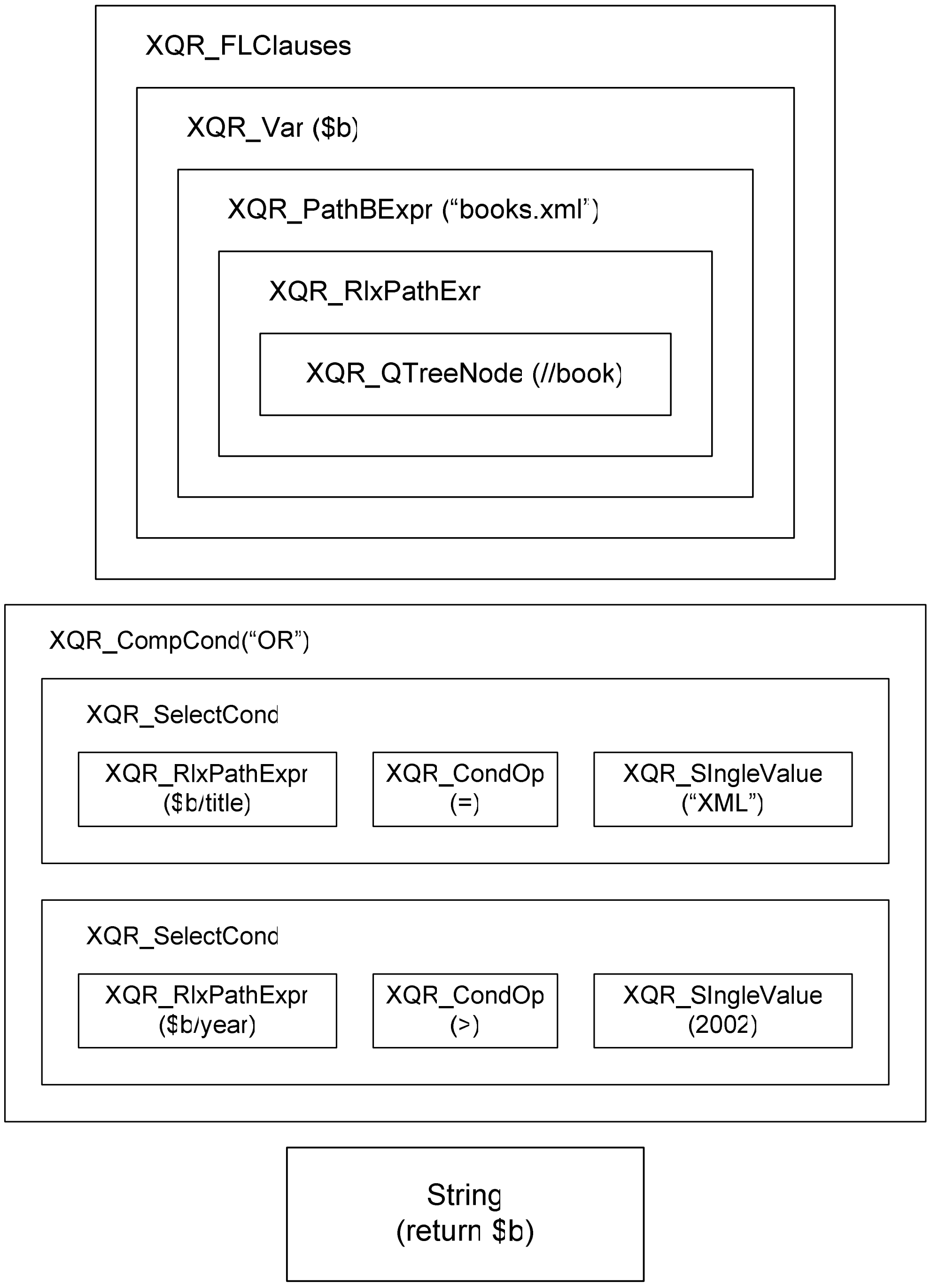


*Figure 17: Example XQueryRep object*

*Figure 18* shows the general framework of the RLXQuery Process. Note that the XQueryRep object is passed between modules during the relaxation process. As modules relax the query, the XQueryRep object will be updated. In addition, the Execution Trace will log all changes to the object.

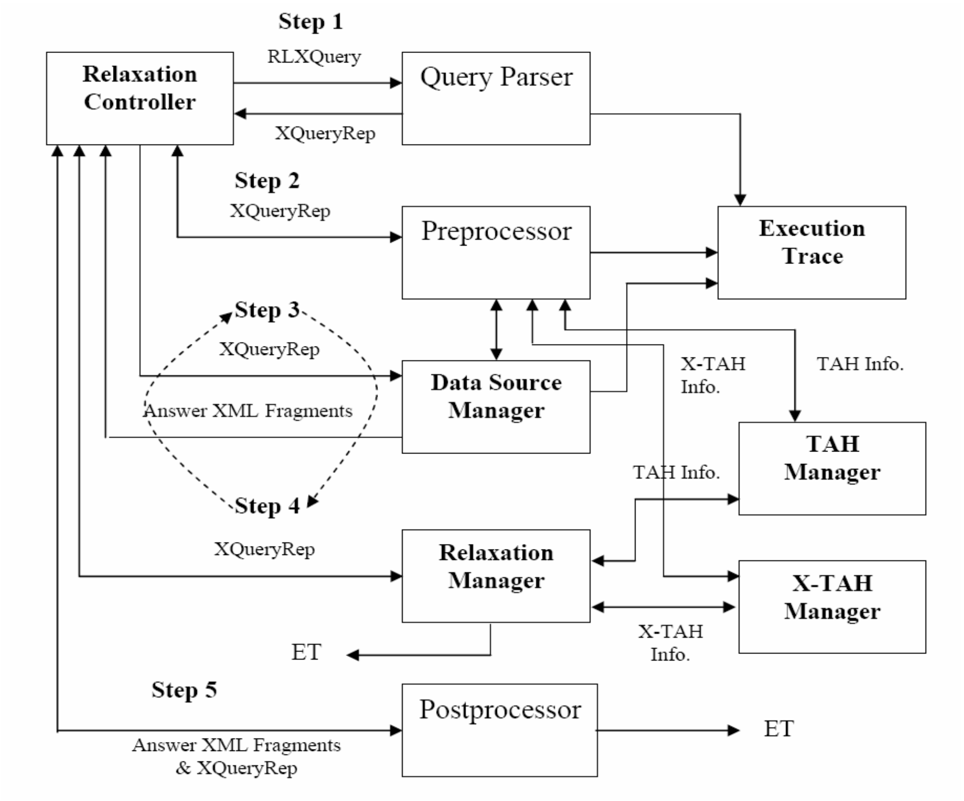


***Figure 18: Overall flow of the RLXQuery query processing***

### 4.2.1.1 The Relaxation Controller

The Relaxation Controller is the main driving module in the RLXQuery Process. Given a set of queries, the Relaxation Controller will control the flow of relaxation for each query. It operates the following steps when answering each XML query:

**Step 1:** Contact the Query Parser to get the XQueryRep object
**Step 2:** Send the XQueryRep object to the preprocessor for some processing
**Step 3:** Send the XQueryRep object to the Data Source Manager to be executed

**Step 4:** Send the XQueryRep object to the Relaxation Manager to be relaxed, repeat Step 3 until enough answers are returned, or the query cannot be relaxed any further.

**Step 5:** Send the XQueryRep to the Postprocessor

#### 4.2.1.2 The Query Parser

The Query Parser is the module responsible for translating the XQuery String to an XQueryRep object. This module will parse the XQuery String and build each component of the XQueryRep object such as the binding expression, the where condition, etc. In addition, the Parser is responsible for checking the validity of the query syntax. This syntax checking is based on the RLXQuery EBNF (see Appendix). The object constructed by the Query Parser is the central object used to analyze and relax the query. Once it is fully constructed, this object will be used as the primary source for finding and performing query relaxations.

#### 4.2.1.3 The Preprocessor

After the XQueryRep object is constructed by the parser, the Relaxation Controller will send it to the Preprocessor. The Preprocessor is responsible for converting some of the RLXQuery constructs within the query such as approximate operators, conceptual terms, SIMILAR-TO constructs, and REJECT constructs. If found, the Preprocessor will convert these RLXQuery constructs into their XQuery equivalents. Once the Preprocessor has finished, the modified XQueryRep object is ready for the further relaxations by the Relaxation Manager.

#### 4.2.1.4 The Relaxation Manager

After being sent to the Preprocessor, the XQueryRep object is sent to the Relaxation Manager for relaxation. Upon receiving the XQueryRep object, the Relaxation Manager will search the query for relaxable conditions and build a relaxation specification accordingly. After this specification has been constructed, it can be used to guide the relaxation of the query. Each relaxation of the query will involve relaxing one of the units found by the Relaxation Manager until enough answers have been returned or there are no more units to relax. We will discuss the Relaxation Manager and its associated functions in detail in the next section.

#### 4.2.1.5 The TAH and X-TAH Managers

The TAH and X-TAH Managers are interfaces used by the RLXQuery engine to obtain value and structure relaxations respectively. Each structure is a hierarchical representation of the data as described in Section 4.1.2. These interfaces allow the RLXQuery modules to submit a value or structure expression and receive a list of relaxed expressions.

#### 4.2.1.6 The Execution Trace

The Execution Trace is used to track changes to the XQueryRep object. During both the Preprocessing and the Relaxation phases the XQueryRep may be modified to reflect relaxation in the query. This information is traced by the Execution Trace Manager by providing an interface for all modules to write messages. Each message is given a unique id and the module writing the message is recorded. In this way, the history of the query relaxation can be traced back in time for debug purposes, or as information to the user executing the query.

#### 4.2.1.7 The Postprocessor

After the query relaxation process has finished, the XQueryRep object and the set of query answers are sent to the Postprocessor. The Preprocessor will rank the answers as specified in the RANK-BY clause in the query. The result of the Postprocessor is an ordered set of answers to the given query. This result set will be returned to the user.

## 5 The Relaxation Manager

This section will focus on the Relaxation Manager, the module responsible for building the relaxation specification and controlling query relaxations. To build the relaxation specification, the Relaxation Manager will search the XQueryRep object for relaxation units, grouping them by common TAH or XTAH.

The Relaxation Manager is a module developed in Java for use in the CoXML system. The Relaxation Manager and its associated objects were implemented in approximately 2000 lines of code. This work builds off the work by Shaorong Liu and Christian Cardenas in the CoXML and Preprocessor respectively.[1]

For each RLXQuery, there is one associated Relaxation Manager that will guide the relaxation process for the entire query. *Figure 19* shows the object hierarchy for the Relaxation Manager. We will describe each of these objects in detail.

[1] References and code can be found on the CoXML website: http://www.cobase.cs.ucla.edu/projects/CoXML/coxml.html

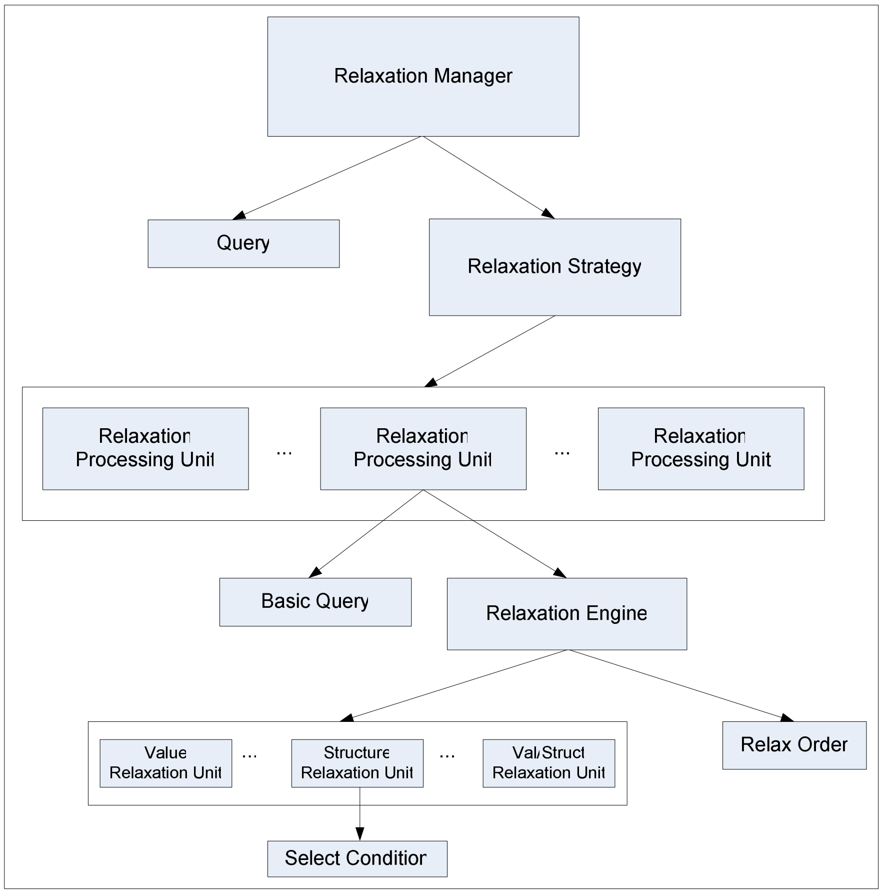


***Figure 19: The Relaxation Manager Object diagram***

## 5.1 The Relaxation Manager

The Relaxation Manager is the top most object responsible for query relaxation. Using the Relaxation Strategy object, the relaxation manager will relax the object with the method *relax()*. This method will then call the *relax()* method of the Relaxation Strategy Object which will relax the Query based on its specifications.

## 5.2 The Relaxation Strategy

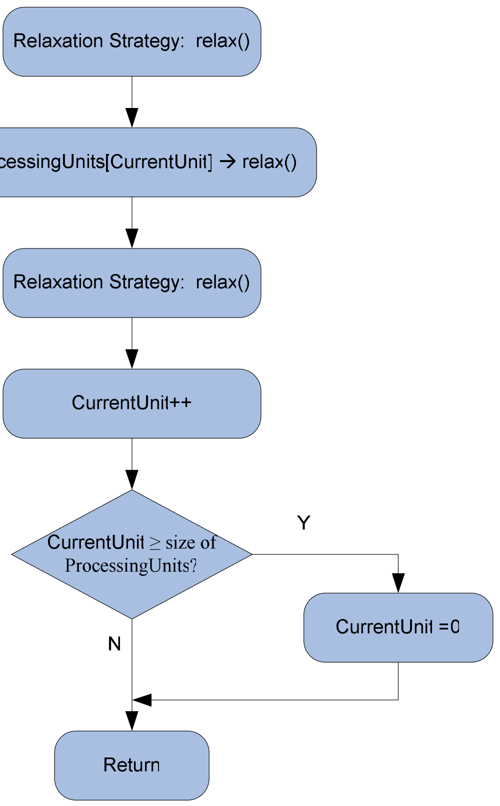
Each Relax Manager has an associated Relaxation Strategy. This strategy is the top-most specification of how to relax the RLXQuery. The Relaxation Strategy contains one or more Relaxation Processing Units, each of which corresponds to a single RLXBasic Query.

We may have multiple Processing Units because a general RLXQuery can be composed of several single RLXQuery queries connected using ‘INTERSECT’, ‘UNION’, or ‘EXCEPT’. For example, consider the RlxQuery:

```
for $b in document("bib.xml")//article
        where $b/year > 2000 and $b/title =~ "XML"
        return $b
UNION
for $c in document("dblp.xml")//article
        where $c/year > 2000 and $c/title =~ "XML
        return $c
```

This query will be split into two Processing Units, one for each Basic Query connected by ‘UNION’. The list of Processing Units is kept in a vector. In addition, a pointer is kept to the Processing Unit currently being relaxed.

Because a RLXQuery is relaxed in steps, the Relaxation Strategy must relax each Processing Unit equally. This is done by implementing a ring for the Vector of Processing Units. During relaxation, the first element of the Vector will be relaxed first, and the second will be relaxed next. After the second element is relaxed, the Relaxation Strategy will wrap around and relax the first Processing Unit further. This process will continue until enough answers have been returned to the user or until no further relaxation can be done. Figure 20 shows the control flow of the relaxation algorithm for the Relaxation Strategy.



Relaxation Strategy: relax()
ProcessingUnits[CurrentUnit] → relax()
Relaxation Strategy: relax()
CurrentUnit++
CurrentUnit ≥ size of ProcessingUnits?
Y
CurrentUnit =0
N
Return

***Figure 20: Relax Algorithm for the Relaxation Strategy***

The Relaxation Strategy also contains a method for determining whether a query is relaxable at the current time. This method, *get_rlx_flag()*, will return TRUE if any part of the query is still relaxable, and FALSE if the query cannot be relaxed any further. *Figure 21* shows the algorithm used in the *get_rlx_flag()* method.

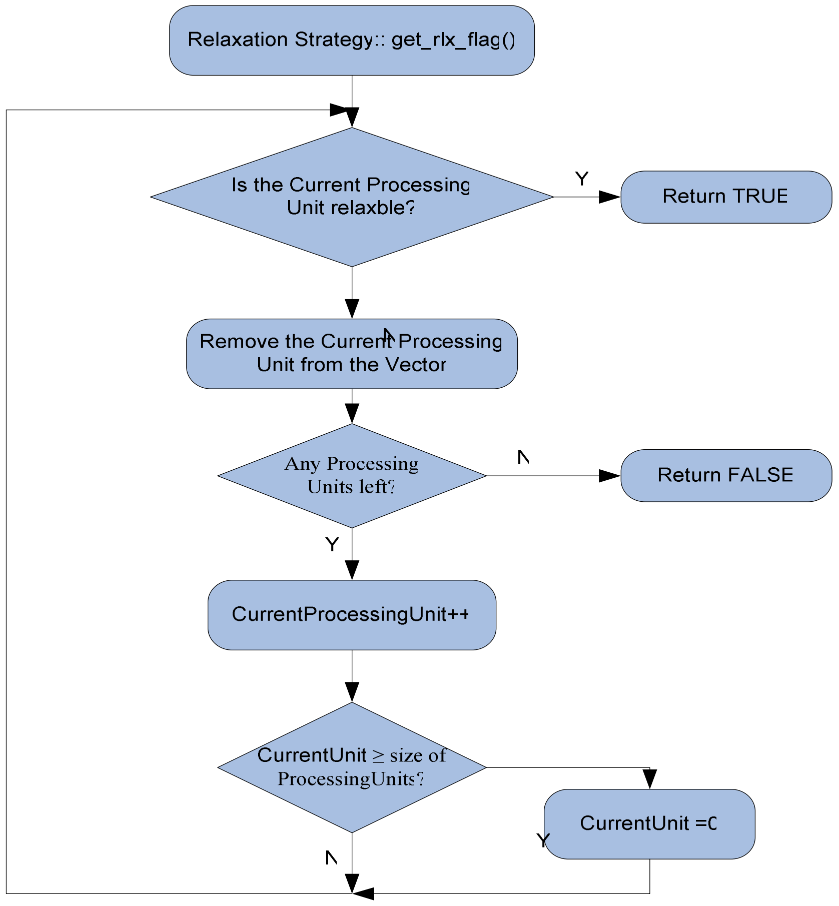

Relaxation Strategy:: get_rlx_flag()
Is the Current Processing Unit relaxble?
Y
Return TRUE
Remove the Current Processing Unit from the Vector
Any Processing Units left?
N
Return FALSE
Y
CurrentProcessingUnit++
CurrentUnit ≥ size of ProcessingUnits?
CurrentUnit =0
Y
N

***Figure 21: Algorithm for get_rlx_flag() in the Relaxation Strategy***

## *5.3 Relaxation Processing Units*

As mentioned earlier, a Relaxation Processing Unit is built for each Basic Query. One exception to this is queries that appear on the right-hand-side of 'EXCEPT'. In this case, the size of the answer set will only decrease. For all other Basic Queries, Relaxation Processing Units will be built to handle the relaxation for its Basic Query.

Each Processing Unit holds a single Basic Query, and its associated Relaxation Engine. After the Relaxation Engine is built, this unit can be relaxed according to the specifications found by the Engine. This includes which conditions to relax as well as the order in which to relax them.

## 5.4 The Relaxation Engine

The Relaxation Engine is responsible for building and controlling the relaxation of its Processing Unit. The Relaxation Engine is the relaxation specification for each single Basic Relax Query.

One of the responsibilities of the Relaxation Engine is to determine the relaxation order of its Processing Unit. During relaxation, the Relaxation Unit will be relaxed in steps, each called a 'relation step'. The relaxation order dictates which relaxation unit(s) will be relaxed at each relaxation step. There are three basic kinds of relaxation order that can be found in the "RELAX-ORDER" clause of a RLXQuery: sequential, simultaneous, and system-defined. Suppose $G_i$ is the group of relaxation units corresponding to the $i^{th}$ relaxation order element ($1 \leq i \leq n$).

- Sequential: specified by "[$G_1$,$G_2$,…$G_n$]". $G_1$,$G_2$,…,$G_n$ are relaxed in sequential order.
- Simultaneous: specified by "($G_1$,$G_2$,…$G_n$)". $G_1$,$G_2$,…,$G_n$ are relaxed simultaneously.
- System-defined: by "{$G_1$,$G_2$,…$G_n$}". $G_1$,$G_2$,…,$G_n$ are relaxed according to a system defined relaxation order.

The Relaxation Engine's main function is to search for relaxation units in the Basic Relax Query it corresponds to. During relaxation, the major part of a Query that will be relaxed is the <cooperative search condition> part, i.e. the part founding the WHERE clause. A <cooperative search condition> will be composed of several <cooperative search Boolean primary condition>s connected by "OR", "AND" or "NOT".

In the XQueryRep object, the WHERE condition is represented as a tree: with the Boolean operators as internal nodes and Select Conditions as leaves. The Select Conditions can be grouped using the Query's where condition tree based on their common TAHs or XTAHs.

To search the XQueryRep's condition tree, the Relaxation Engine begins by finding starting points in the tree from which to search.  A starting point is defined as follows:

1. If the root node of the entire condition tree corresponds to an "AND" operator, then it is a starting point
2. The root nodes of the two branches that act as two operands of an "OR" operator are starting points.

The Relaxation Engine finds starting points in the tree by recursively descending down the where condition tree.  If one of the conditions of a starting point is met, the Condition is added to the list of starting points, otherwise the algorithm will continue to search the tree.

For example, the starting points for the tree shown in *Figure 22*(a) will have the starting points (marked) shown in *Figure 22*(b).

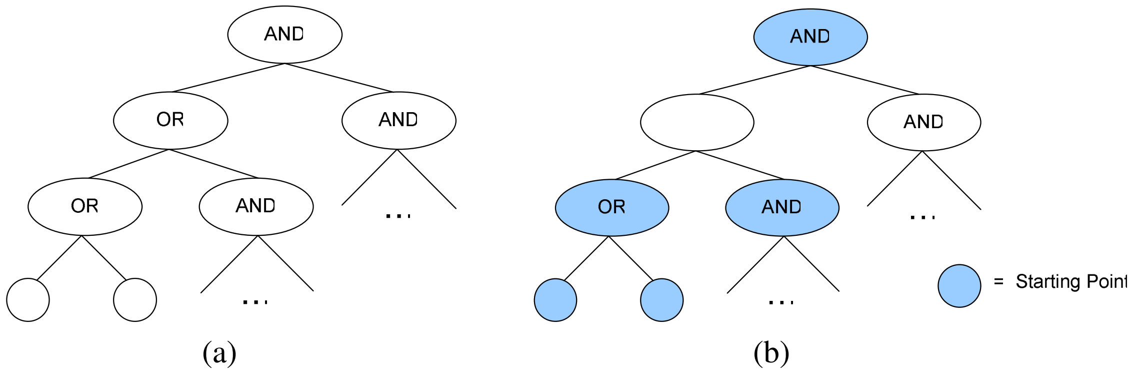


***Figure 22: Sample Condition Tree and Corresponding Starting Points***

Using these starting points, the Relaxation Engine will search the where condition tree for relaxation units. The starting points can be thought of as ranges to be searched since we are only interested in finding leaf nodes continuously connected by "AND". These searching ranges will not overlap. The searching ranges are shown in *Figure 23*.

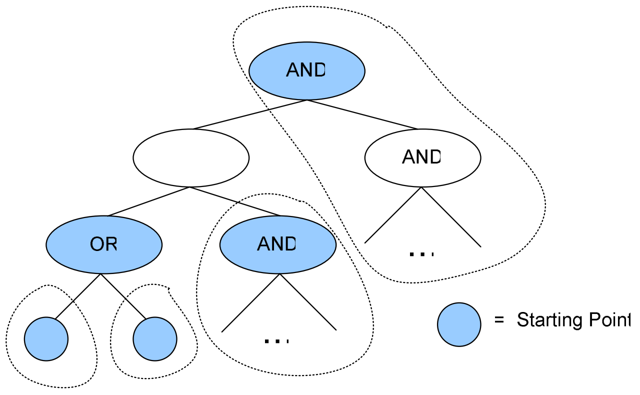


***Figure 23: Searching range represented by each starting point***

To search for relaxation units, we search beginning at each starting point. Starting from the branch, whenever a node (branch or leaf) associated with a <tah alias level clause> is found we group all the leaves:

1. in the branch starting from that node, and
2. contain path expression from the same document

Whenever a node (branch or leaf) associated with a <xtah alias level clause> is found, we group all the leaves:

1. in the branch starting from that node, and
2. contain path expression from the same document

The process of searching for relaxation units is done using a Hash Table with:

- Key = the TAH/XTAH label plus the document root common to the group
- Value = list of Selection Conditions for the label

To determine whether path expressions are from the same document, we must traverse the path expression tree for each expression and find the root binding expressions for these path expressions. In addition, path expressions can be directly or indirectly defined from other path expressions. In this case, we must find up top-most binding expression to which this path expression belongs. For example, consider the following query:

```
for $c in document(“collections.xml”)/collection,
   $b in $c/book
return $b
```

Here we see that $b is directly defined from $c, and both expression belong to the “collections.xml” document. In general, we may have a series of variables defined by other variables $\{v_1,v_2,\ldots, v_n\}$ in which $v_i$ is directly defined from $v_{i+1.}$ This series of variable must be traversed to determine the top-most binding root.

*Figure 24* below shows the algorithm for the *search()* method for finding relaxation units in the where condition tree. As groups are found, the proper key is composed and the (key, value) pair is put into the Hash Table.

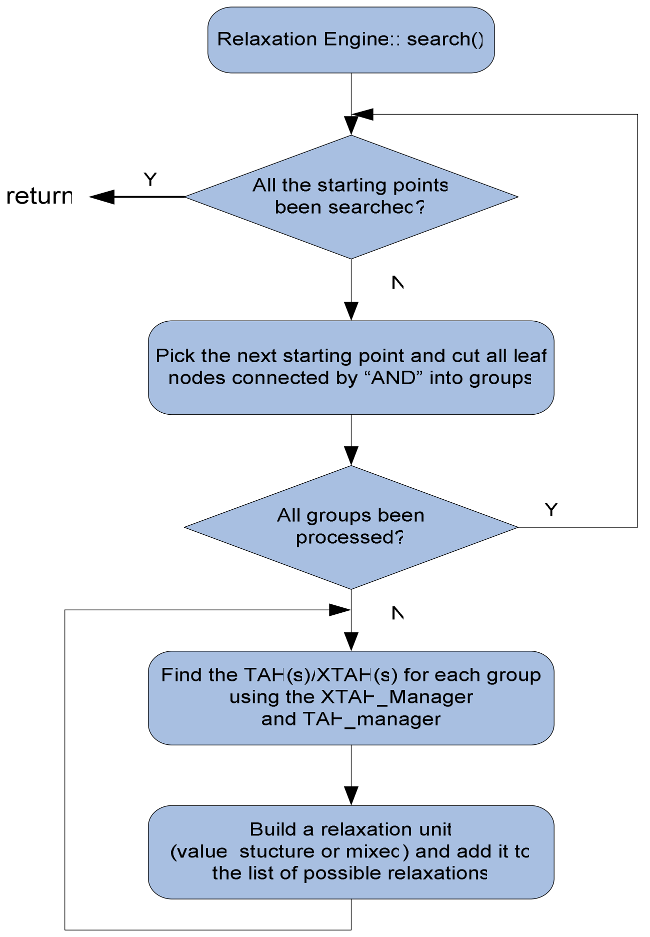

Relaxation Engine:: search()
All the starting points been searched?
Y
return
N
Pick the next starting point and cut all leaf nodes connected by “AND” into groups
All groups been processed?
Y
N
Find the TAH(s)/XTAH(s) for each group using the XTAH_Manager and TAH_manager
Build a relaxation unit (value stucture or mixed) and add it to the list of possible relaxations

***Figure 24: Algorithm for search() method of the Relaxation Engine***

Relaxation units can also be found in the FOR/LET clause of the Query. In this case, the binding expression may be relaxable. The engine will also search this clause for relaxation units.

After all relaxation units have been found in the WHERE condition and in the FOR/LET clause, the Relaxation Engine maps these relaxation units to the relaxation order. This is done by composing an ordered list of Relaxation Units based on the order given in the RELAX-ORDER clause. Relaxation units that are not referred to by the user are always relaxed after relaxation units that are referred to in the RELAX-ORDER clause. In addition, these non-referred to units are relaxed using the "system-defined" relaxation order.

### 5.4.1 Basic Relaxation Units – Value, Structure, and Mixed

The groups found in the Relaxation Engine's search process are denoted 'Basic Relaxation Units' and can be one of three types: a value relaxation unit, a structure relaxation unit or a mixed relaxation unit.

1. **Value relaxation unit**: for value relaxations. A value relaxation unit can be either:
    1) One <cooperative predicate> associated with a single TAH, or
    2) A group of <cooperative predicate>s associated with a multiple TAH
2. **Structure relaxation unit**: for structure relaxations. A structure relaxation unit can be either:
    1) One <cooperative predicate> associated with a single XTAH, or
    2) A group of <cooperative predicate>s associated with a multiple XTAH, or
    3) A variable binding expression that is not-relaxable. A variable's binding expression is non-relaxable if it is explicitly specified with a <NON-RELAXABLE Path Expression>.
3. **Mixed relaxation unit**: for units that has both value and structure relaxations.

If a variable is a relaxation order element in the RELAX-ORDER clause, then its corresponding binding expression is a structure relaxation unit.

Each Basic Relaxation Unit(either value, structured or mixed) is implemented as a sub-class of a Basic Relaxation Unit. These relaxation units will be built for each group found in the where condition tree by the Relaxation Engine. For each group, the Relaxation Engine will determine which type of relaxation unit to build based on the relaxation condition(s) given by the user.

Each sub-class of the Basic Relaxation Unit has a *relax()* method capable of relaxing the selection condition(s) it represents. If the TAH/X-TAH is specified, the *relax()* method will contact the appropriate TAH and/or X-TAH and get a list of relaxations for the condition(s). The XQueryRep will then be updated with this new value and the change will be logged in the Execution Trace Manager. If the TAH/X-TAH is not specified, the *relax()* method will first find an appropriate TAH/X-TAH to use, and then continue the same. In this way, the Relaxation Engine can simply direct a Basic Relaxation Unit to relax without being concerned with the details of the particular unit's relaxation.

For example, consider the Basic Query:

```
for $b in document("bib.xml")//book
where $b/~author = "Anna" USE-XTAH t1 OR
  ($b/title =~ "XML" AND $b/price =~ "50") USE-TAH t2
return $b
RELAX-ORDER [t2,t1]
```

The where condition for this query can be represented as the following tree:

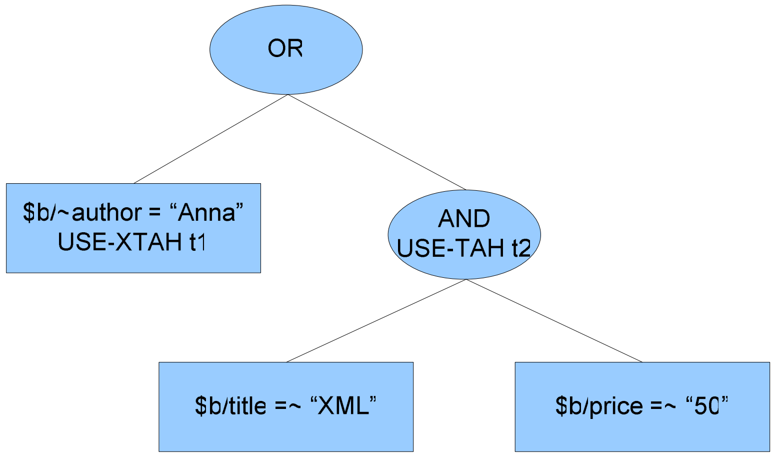


***Figure 25: Example where condition tree***

The Relaxation Engine for this query builds Basic Relaxation Units for each common group of conditions. For the previous example, the leaves will be grouped as follows:

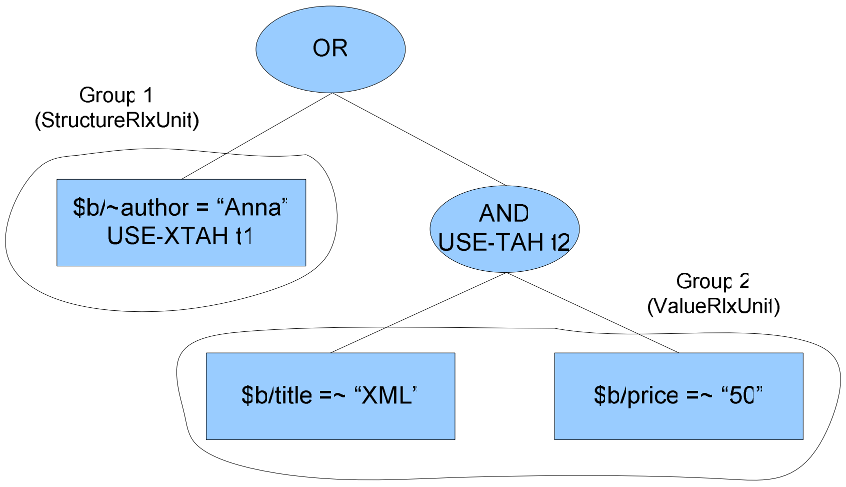


***Figure 26: Example Basic Relaxation grouping***

Because the relaxation order is specified as "[t2,t1]", sequential ordering will be used with Group 2 being relaxed first followed by the relaxation of Group 1.

For the relaxation of Group1, the node “author” will be relaxed. For this relaxation the Structure Relaxation Unit will contact the XTAH Manager and ask for relaxations for “book/author” using X-TAH “t2”. An example result set for this relaxation may be (“book/by”, “book/name”, and “book/auth”). The Structure Relaxation Unit will then modify the XQueryRep object so that the new, relaxed query can be executed by the Relaxation Controller. In addition, a message will be written to the Execution Trace Manager indicating the how the query was relaxed, the old path expression, and the new path expression.

For the relaxation of Group 2, both Select conditions (“$b/title =~ “XML” and “$b/price =~ “50”) will be relaxed. In this case, the Value Relaxation Unit will contact the TAH Manager and ask for relaxations for “book/title” and “book/price” using TAH “t1”. An example result set for this relaxation may be (“Semi-Structured Data”) for “book/title” and the range 40-70 for “book/price”. Like with the Structure Relaxation Unit, the Value Relaxation unit will then modify the XQueryRep object with the new values and write appropriate messages to the Execution Trace Manager.

The following shows the evolution of the Query as relaxations are performed:

Original Query:

```
for $b in document(“bib.xml”)//book
where $b/~author = “Anna” USE-XTAH t1 OR
  ($b/title =~ “XML” AND $b/price =~ “50”) USE-TAH t2
return $b
RELAX-ORDER [t2,t1]
```

After the 1st Relaxation (value relaxation):

```
for $b in document(“bib.xml”)//book
where $b/~author = “Anna” USE-XTAH t1 OR
  ($b/title = “Semi-Structured Data” AND
  $b/price = [40-70]) USE-TAH t2
```

```
return $b
RELAX-ORDER [t2,t1]
```

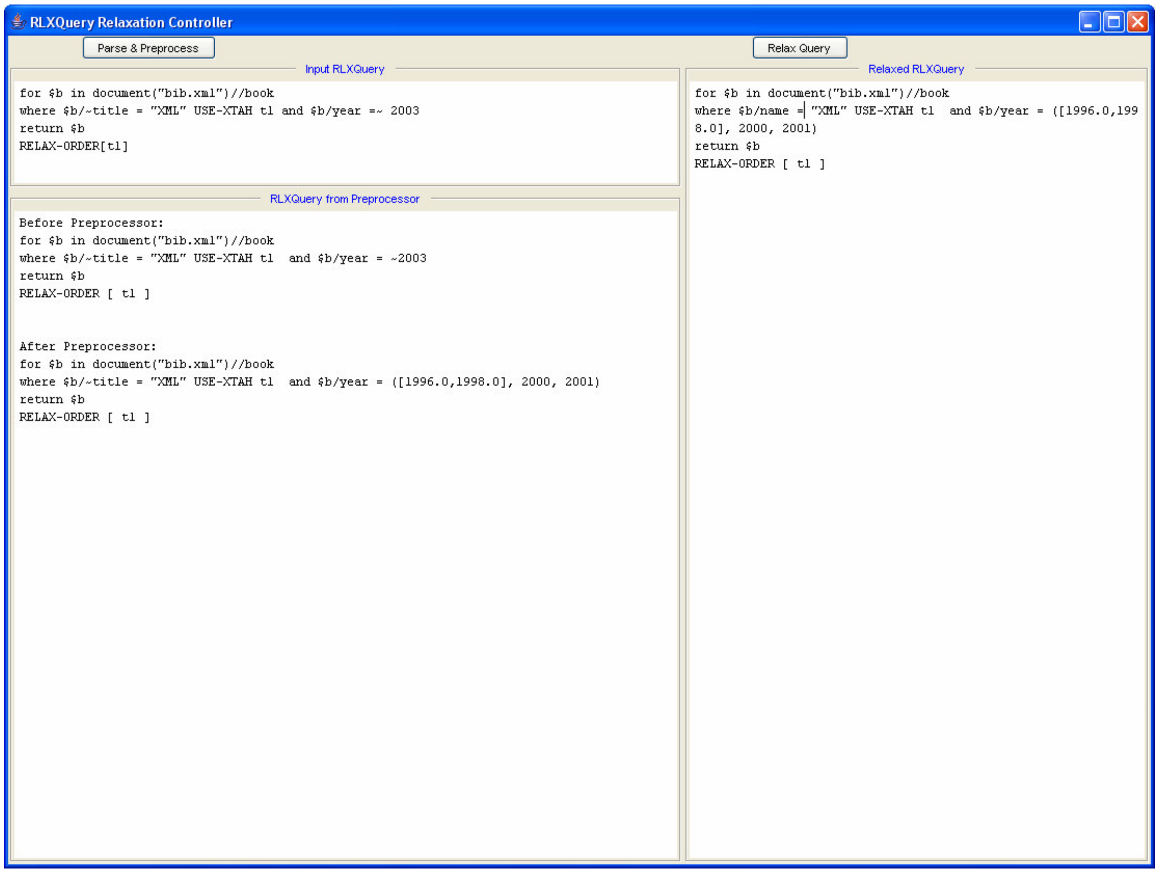
After the 2nd Relaxation (structure relaxation)

```
for $b in document(“bib.xml”)//book
where ($b/by = “Anna” OR $b/name = “Anna” OR
 $b/auth = “Anna”) USE-XTAH t1 OR
 ($b/title = “Semi-Structured Data” AND
 $b/price = [40-70]) USE-TAH t2
return $b
RELAX-ORDER [t2,t1]
```

After this second relaxation, no further relaxations can be performed. If the user attempts to relax the query further, the Relaxation Strategy will find the *get_rlx_flag()* returns FALSE and will report that the query can not be relaxed any further.

## 5.5 The Relaxation GUI

*Figure 27* below shows the layout of the GUI used for relaxation. In the top left pane, a user can submit a query to be relaxed. When the “Parse and Preprocess” button is clicked the Query will be sent to the Parser and then the Preprocessor. If the query entered is invalid an appropriate error message is given. If the query is valid, the left bottom pane will display the query both before and after preprocessing. Once the Query has be parsed and preprocessed, the user can use the “Relax Query” button to request that the query be relaxed one step. The user can relax the query multiple times and is given a message indicating that query cannot be relaxed any further. After each relaxation, the new, modified query is shown.



```
for $b in document("bib.xml")//book
where $b/~title = "XML" USE-XTAH tl and $b/year =~ 2003
return $b
RELAX-ORDER[tl]
```

```
Before Preprocessor:
for $b in document("bib.xml")//book
where $b/~title = "XML" USE-XTAH tl  and $b/year = ~2003
return $b
RELAX-ORDER [ tl ]


After Preprocessor:
for $b in document("bib.xml")//book
where $b/~title = "XML" USE-XTAH tl  and $b/year = ([1996.0,1998.0], 2000, 2001)
return $b
RELAX-ORDER [ tl ]
```

```
for $b in document("bib.xml")//book
where $b/name = "XML" USE-XTAH tl  and $b/year = ([1996.0,199
8.0], 2000, 2001)
return $b
RELAX-ORDER [ tl ]
```

***Figure 27: Screen Shot of the Relaxation GUI***

In the example shown, the original query is:

```
for $b in document(“bib.xml”)//book
where $b/~title = “XML” USE-XTAH t1 and $b/year =~2003
return $b
RELAX_ORDER[t1]
```

After Parsing and Preprocessing, the approximate year is relaxed and the query becomes (show in the lower, left pane):

```
for $b in document(“bib.xml”)//book
where $b/~title = “XML” USE-XTAH t1 and $b/year =
   ([1996.0, 1998.0], 2000, 2001)
return $b
RELAX_ORDER[t1]
```

After Relaxation, the node “title” is relaxed to “name” and the query becomes (shown in the right pane):

```
for $b in document(“bib.xml”)//book
where $b/name = “XML” USE-XTAH t1 and $b/year =
   ([1996.0, 1998.0], 2000, 2001)
return $b
RELAX_ORDER[t1]
```

After this relaxation, the query cannot be relaxed any further.

# 6 Conclusion

The popularity of XML as an extensible document language makes accessing its data of paramount importance. Moreover, the complex nature of XML as well as the wealth of heterogeneous XML information sources makes XML query relaxation an important component to any XML query answering system. By including query relaxation capabilities, an XML query answering system can provide approximate answers when the number of exact matches is not satisfactory.

In this paper, we explored various aspects of XML query answering. We began by discussing the XML data model itself as well as traditional XML query languages. We then examined relaxation types including value relaxations and structural relaxations. We also briefly discussed relaxations controls; options given to the user to specify how query relaxations should be performed.

We also looked at CoXML[1]; as system attempting to implement a cooperative XML querying system that includes XML query relaxation. In particular, we described the design and function of the Relaxation Manager module. This module is responsible for identifying and controlling the relaxation of an XML query in the CoXML system. We showed some of the implementation details of this module including the objects used and their associate properties and methods.

There is still much to be done in the area of XML query relaxation. The CoXML project is currently working toward providing a fully functional XML Query Relaxation language and query processing system utilizing domain information captured in TAHs and X-TAHs. The Relaxation Manager allows the system to build relaxation specifications for queries, as well as control the relaxation process. Future work includes further expanding the functionality of the Relaxation Manager to handle larger and more complex queries.

# Appendix A: RLXQuery EBNF

| | | |
|---|---|---|
| RLXQuery | ::= | (FunctionDefn)* RLXQuerySpecification |
| RLXQuerySpecification | ::= | RLXQuerySpecItem ((“union” \| “intersect” \| “except”) RLXQuerySpecItem)* |
| RLXQuerySpecItem | ::= | RLXFLWORExpr \| “(“ RLXQuerySpecification “)” |
| RLXFLWORExpr | ::= | (RLXForClause \| RLXLetClause)$^{+}$ (RLXWhereClause)? (OrderByClause)? ReturnClause (RelaxationOrderClause \| AtLeastClause \| RankMethodClause \| USEClause )* |
| RLXForClause | ::= | <ForVariable> <VarName> <In> BExpr (<Comma> <VariableIndicator> <VarName> <In> BExpr)* |
| BExpr | ::= | NonRelaxablePathExpr \| SpecialBExpr \| Expr |
| | | |
| SpecialBExpr | ::= | (<Root>SpecialRelativeBExpr?) \| (<RootDescendant>? SpecialRelativeBExpr) |
| SpecialRelativeBExpr | ::= | SpecialBStepExpr (<NonRelaxable>? (<Slash>\|<SlashSlash>) SpecialBStepExpr)* |
| SpecialBStepExpr | ::= | ( SimpAbbrevForwardStep\|(“$” VarName) \|( “document(“ StringLiteral“)”) ) (“[” BPredicateExpr “]”)* |
| SimpAbbrevForwardStep | ::= | (ApproximateNode \| NonRelaxableNode \| (“@”? (QName\| Wildcard)) |
| | | |
| BPredicateExpr | ::= | BPredicateTerm (“and” BPredicateTerm)* |
| BPredicateTerm | ::= | (“contains(“ PredicatePathExpr “,” StringLiteral”)”) \|(“not” “(“ “contains” “(“ PredicatePathExpr “,” StringLiteral “))” \| (PreidcatePathExpr BPredicateCompOpElem?) |
| BPredicateCompOpElem | ::= | (“=” (ExactValue))\| ((“<” \| “>” \| “<=” \| “>=” \|”!=”) Literal) |
| PredicatePathExpr | ::= | (“.” (“!”? (“/”\|”//”) PredicateRelativePathExpr)? )\| PredicateRelativePathExpr |
| PredicateRelativePathExpr | ::= | SimpAbrevForwardStep ((“!”)? (“/” \| “//”) SimpAbbrevForwardStep)? |
| ApproximateNode | ::= | “~” @? QName |
| NonRelaxableNode | ::= | <NonRelaxable> @? QName |
| | | |
| RLXLetClause | ::= | <LetVariable> <VarName> “:=” BExpr (<Comma> <VariableIndicator> <VarName> “:=”BExpr)* |
| RLXWhereClause | ::= | “where” RLXWhereExpr |
| RLXWhereExpr | ::= | RLXAndExpr (“or” RLXWhereExpr)* |
| RLXAndExpr | ::= | RLXCompExpr (“and” RLXAndExpr)* |
| | | |
| RLXCompExpr | ::= | ((“(“ (CooperativePredicate \| ComparisonExpr) ( (“,” SimplePathExpr)+ “)” SIMILAR-TO “(“ GeneralValueList “)” BasedOnClause?) \| (“Or” RLXAndExpr “)”) \| (“And” RLXCompExpr “)”)) \| CooperativePredicate \| ComparisonExpr ) |

| | | |
|---|---|---|
| | | XTahAliasLevelClause? TahAliasLevelClause? |
| TahAliasLevelClause | ::= | UseTahClause ((VConditionLabel VRelaxationLevel?) \| (VRelaxationLevel VConditionLabel?))? \|VConditionLabel ((VRelaxationLevel UseTahClause?) \| (UseTahClause VRelaxationLevel?))? \|VRelaxationLevel ((VConditionLabel UseTahClause?) \| (UseTahClause VConditionLabel?))? |
| UseTahClause | ::= | <UseTah> QName |
| VConditionLabel | ::= | <VCondAlias> QName |
| | | |
| VRelaxationLevel | ::= | <RelaxLevelV> NumericLiteral |
| | | |
| XTahAliasLevelClause | ::= | UseXTahClause (( SConditionLabel SRelaxationLevel?) \| ( SRelaxationLevel SConditionLabel?))? \|SConditionLabel ((UseXTahClause SRelaxationLevel?) \| (SRelaxationLevel \| UseXTahClause)?)? \|SRelaxationLevel (SConditionLabel UseXTahClause?) \|(UseXTahClause SConditionLabel?))? |
| UseXTahClause | ::= | <UseXTah> QName |
| SConditionLabel | ::= | <SCondAlias> QName |
| SRelaxationLevel | ::= | <RelaxLevelS> NumericLiteral |
| | | |
| CooperativePredicate | ::= | CooperativeCount \| CooperativeCompPredicate |
| | | |
| SimplePathExprList | ::= | "(" SimplePathExpr ("," SimplePathExpr)* ")" |
| SimplePathExpr | ::= | ("/" SimpleRelativePathExpr?) \| ("//" SimpleRelativePathExpr) \| SimpleRelativePathExpr |
| SimpleRelativePathExpr | ::= | SimpleStepExpr (("/" \| "//") SimpleStepExpr)* |
| SimpleStepExpr | ::= | "$"VarName \| (@? QName) |
| | | |
| GeneralValueList | ::= | (StringLiteral \| GeneralNumeriElem) (<Comma> (StringLiteral \| GeneralNumericElem))* |
| GeneralNumericElem | ::= | NumericLiteral \| NumericRange |
| | | |
| BasedOnClause | ::= | "BASED-ON" <Lpar> BasedOnPathExprList <Rpar> |
| BasedOnPathExprList | ::= | (SimplePathExpr ("," SimplePathExpr)*) \| (PathExprWeightElem ("," PathExprWeightElem)*) |
| PathExprWeightElem | ::= | <Lpar> SimplePathExpr <Comma> NumericLiteral <Rpar> |
| | | |
| CooperativeCompPredicate | ::= | CooperativePathExpr CompOpElem? |
| CooperativeCount | ::= | "count(" SimplePathExpr ")" ( ("=" (CooperativeValueElem \| GeneralNumericElem\| ValuePreferenceList ValueRejectionList?)) \| ((">" \| ">=" \| "<" \| "<=" \| "!=") (CooperativeValueElem \| |

| | | |
|---|---|---|
| | | NumericLiteral)) |
| CompOpElem | ::= | ("=" (CooperativeValueElem \| ExactValue \| ValuePreferenceList ValueRejectionList?)) \| (("\>" \| "\>=" \| "\<" \|"\<=" \| "!=") (CooperativeValueElem \| ExactValue)) \| (SingleSimilarToElem BasedOnClause?) |
| ExactValue | ::= | Literal \| NumericRange |
| NumericRange | ::= | \<Lbrack\> NumericLiteral \<Comma\> NumericLiteral \<Rbrack\> |
| CooperativePathExpr | ::= | NonRelaxablePathExpr \| SpecialPathExpr |
| NonRelaxablePathExpr | ::= | "!" "(" SimplePathExpr ")" |
| SpecialPathExpr | ::= | \<Root\> SpecialRelativePathExpr? \| \<RootDescendants\> SpecialRelativePathExpr \| SpecialRelativePathExpr |
| SpecialRelativePathExpr | ::= | SpecialStepExpr ((\<NonRelaxable\>)? (\<Slash\> \| \<SlashSlash\>)) SpecialStepExpr)* |
| SpecialStepExpr | ::= | (SimpAbbrevForwardStep \| ("$" VarName) \| SpecialFunctionCall) SpecialPredicates |
| SpecialPredicates | ::= | ("[" SpecialPredicateExpr "]" )* |
| SpecialPredicateExpr | ::= | SpecialPredicteTerm ("and" SpecialPredicateTerm)* |
| | | |
| SpecialPredicateTerm | ::= | ( "contains(" PredicatePathExpr "," StringLiteral ")" ) \| ( PredicatePathExpr PredicateCompOpElem?) \| |
| PredicateCompOpElem | ::= | ("=" (CooperativeValueElem \| ExactValue)) \| (("\<" \| "\>" \| "\<=" \| "\>=" \|"!=") (CooperativeValueElem \| Literal )) |
| SpecialFunctionCall | ::= | "contains" "("SpecialPathExpr "," ((StringLiteral \| ValuePreferenceList) (ValueRejectionList)?) ")" \| "document" "(" StringLiteral ")" |
| SingleSimilarToElem | ::= | \<SimilarTo\> ExactValue |
| CooperativeValueElem | ::= | ConceptualValue \| ApproximateValue \| NonRelaxableValue |
| ConceptualValue | ::= | "#" StringLiteral |
| ApproximateValue | ::= | "~" ExactValue |
| NonRelaxableValue | ::= | "!" ExactValue |
| ValuePreferenceList | ::= | \<Prefer\> GeneralValueList |
| ValueRejectionList | ::= | \<Reject\> GeneralValueList |
| | | |
| ReturnClause | ::= | \<Return\> ExprSingle (\<Comma\> ExprSingle)* |
| | | |
| Expr | ::= | ExprSingle (\<Comma\> ExprSingle)* |
| ExprSingle | ::= | FLWORExpr \| QuantifiedExpr \|OrExpr |
| FLWORExpr | ::= | (ForClause \| LetClause)+ WhereClause? OrderByClause? ReturnClause |
| ForClause | ::= | \<ForVariable\> \<VarName\> \<In\> ExprSingle (\<Comma\> \<VariableIndicator\> \<VarName\> \<In\> ExprSingle)* |
| LetClause | ::= | \<LetVariable\> \<VarName\> \<ColonEquals\> ExprSingle (\<Comma\> \<VariableIndicator\> \<VarName\> \<ColonEquals\> |

| | | |
|---|---|---|
| | | ExprSingle)* |
| WhereClause | ::= | “where” Expr |
| OrderByClause | ::= | ( <OrderBy> \| <OrderByStable> ) OrderSpecList |
| OrderSpecList | ::= | OrderSpec ( <Comma> OrderSpec )* |
| OrderSpec | ::= | ExprSingle OrderModifier |
| OrderModifier | ::= | ( ( <Ascending> \| <Descending> ) )? ( ( <EmptyGreatest> \| <EmptyLeast> ) )? ( <Collation> <StringLiteral> )? |
| | | |
| QuantifiedExpr | ::= | (<”some” “$”> \| <”every” “$”>) VarName “in” ExprSingle (“, “ “$” VarName in PathExpr) “satisfies” ExprSingle |
| OrExpr | ::= | AndExpr ( "or" AndExpr )* |
| AndExpr | ::= | ComparisionExpr (“and” ComparisionExpr)* |
| ComparisionExpr | ::= | RangeExpr ((ValueComp \| GeneralComp \| NodeComp \| OrderComp) RangeExpr )? |
| RangeExpr | ::= | AdditiveExpr (“to” AdditiveExpr)? |
| AdditiveExpr | ::= | MultiplicativeExpr ( ("+" \| "-") MultiplicativeExpr )* |
| MultiplicativeExpr | ::= | UnaryExpr ( ("*" \| "div" \| "idiv" \| "mod") UnaryExpr )* |
| UnaryExpr | ::= | ("-" \| "+")* UnionExpr |
| UnionExpr | ::= | IntersectExceptExpr ( ("union" \| "\|") IntersectExceptExpr )* |
| IntersectExceptExpr | ::= | PathExpr ( ("intersect" \| "except") PathExpr )* |
| PathExpr | ::= | ("/" RelativePathExpr?) \| ("//" RelativePathExpr) \| RelativePathExpr |
| RelativePathExpr | ::= | StepExpr (("/" \| "//") StepExpr)* |
| StepExpr | ::= | AxisStep \| FilterStep |
| AxisStep | ::= | (AbbrevForwardStep \| AbbrevReverseStep) Predicates |
| AbbrevForwardStep | ::= | "@"? NodeTest |
| FilterStep | ::= | PrimaryExpr Predicates |
| PrimaryExpr | ::= | Literal \| FunctionCall \| ContextItemExpr \| ("$" VarName) \| ParenthesizedExpr \| Constructor |
| Predicates | ::= | ( “[“ ExprSingle “]” ) |
| Constructor | ::= | ElementConstructor \| XmlComment \| XmlPI \| CdataSection \| CompDocConstructor \| CompElemConstructor \| CompAttrConstructor \| CompNSConstructor \| CompTextConstructor \| CompXmlPI \| ComputedXmlComment |
| ValueComp | ::= | ( <FortranEq> \| <FortranNe> \| <FortranLt> \| <FortranLe> \| <FortranGt> \| <FortranGe> ) |
| NodeComp | ::= | ( <Is> \| <IsNot> ) |
| OrderComp | ::= | ( <LtLt> \| <GtGt> ) |
| GeneralComp | ::= | “<” \| “>” \| “<=” \| “>=” \| “=” \|”!=” |
| Literal | ::= | ( NumericLiteral \| <StringLiteral> ) |
| NumericLiteral | ::= | ( <IntegerLiteral> \| <DecimalLiteral> \| <DoubleLiteral> ) |
| ParenthesizedExpr | ::= | <Lpar> ( Expr )? <Rpar> |
| FunctionCall | ::= | ( <QNameLpar> ) (Expr)? <Rpar> |
| ElementConstructor | ::= | ( <StartTagOpenRoot> \| <StartTagOpen> ) <TagQName> AttributeList ( <EmptyTagClose> \| ( <StartTagClose> |

| | | |
|---|---|---|
| | | ( ElementContent )* <EndTagOpen> <TagQName> ( <S> )? <EndTagClose> ) ) |
| ComputedDocumentConstructor | ::= | ( <DocumentLbrace> Expr <Rbrace> ) |
| ComputedElementConstructor | ::= | ( <ElementQNameLbrace> \| ( <ElementLbrace> Expr <Rbrace> <LbraceExprEnclosure> ) ) ( Expr )? <Rbrace> |
| ComputedAttributeConstructor | ::= | ( <AttributeQNameLbrace> \| ( <AttributeLbrace> Expr <Rbrace> <LbraceExprEnclosure> ) ) ( Expr )? <Rbrace> |
| ComputedTextConstructor | ::= | <TextLbrace> ( Expr )? <Rbrace> |
| CdataSection | ::= | <CdataSectionStart> ( <CDataSectionChar> )* <CdataSectionEnd> |
| XmlProcessingInstruction | ::= | <ProcessingInstructionStart> <PITarget> ( <PIContentChar> )* <ProcessingInstructionEnd> |
| XmlComment | ::= | <XmlCommentStart> ( <CommentContentChar> )* <XmlCommentEnd> |
| ElementContent | ::= | ( <ElementContentChar> \| <LCurlyBraceEscape> \| <RCurlyBraceEscape> \| ElementConstructor \| EnclosedExpr \| CdataSection \| <CharRef> \| <PredefinedEntityRef> \| XmlComment \| XmlProcessingInstruction ) |
| AttributeList | ::= | ( <S> ( <TagQName> ( <S> )? <ValueIndicator> ( <S> )? AttributeValue )? )* |
| AttributeValue | ::= | ( ( <OpenQuot> ( ( <EscapeQuot> \| AttributeValueContent ) )* <CloseQuot> ) \| ( <OpenApos> ( ( <EscapeApos> \| AttributeValueContent ) )* <CloseApos> ) ) |
| AttributeValueContent | ::= | ( <QuoteAttributeContentChar> \| <AposAttributeContentChar> \| <CharRef> \| <LCurlyBraceEscape> \| <RCurlyBraceEscape> \| EnclosedExpr \| <PredefinedEntityRef> ) |
| EnclosedExpr | ::= | ( <Lbrace> \| <LbraceExprEnclosure> ) Expr <Rbrace> |
| | | |
| FunctionDefn | ::= | <DefineFunction> <QNameLpar> ( ParamList )? ( <Rpar> \| ( <RparAs> SequenceType ) ) ( EnclosedExpr \| <External> ) |
| ParamList | ::= | Param ( <Comma> Param )* |
| Param | ::= | <VariableIndicator> <VarName> ( TypeDeclaration )? |
| TypeDeclaration | ::= | <As> SequenceType |
| SingleType | ::= | AtomicType ( <OccurrenceZeroOrOne> )? |
| SequenceType | ::= | ( ( ItemType ( OccurrenceIndicator )? ) \| <Empty> ) |
| AtomicType | ::= | <QNameForSequenceType> |
| ItemType | ::= | ( AtomicType \| KindTest \| <Item> ) |
| KindTest | ::= | ( DocumentTest \| ElementTest \| AttributeTest \| ProcessingInstructionTest \| CommentTest \| TextTest \| AnyKindTest ) |
| ElementTest | ::= | ( <ElementType> \| <ElementTypeForKindTest> ) ( ( ( SchemaContextPath LocalName ) \| ( NodeName ( <CommaForKindTest> TypeName ( <Nillable> )? )? ) ) )? <RparForKindTest> |
| AttributeTest | ::= | ( <AttributeType> \| <AttributeTypeForKindTest> ) ( ( ( SchemaContextPath <At> LocalName ) \| ( <At> NodeName |

| | | |
|---|---|---|
| | | ( <CommaForKindTest> TypeName )? ) )? <RparForKindTest> |
| ProcessingInstructionTest | ::= | ( <ProcessingInstructionLpar> \| <ProcessingInstructionLparForKindTest> ) ( <StringLiteralForKindTest> )? <RparForKindTest> |
| DocumentTest | ::= | ( <DocumentLpar> \| <DocumentLparForKindTest> ) ( ElementTest )? <RparForKindTest> |
| CommentTest | ::= | ( <CommentLpar> \| <CommentLparForKindTest> ) <RparForKindTest> |
| TextTest | ::= | ( <TextLpar> \| <TextLparForKindTest> ) <RparForKindTest> |
| AnyKindTest | ::= | ( <NodeLpar> \| <NodeLparForKindTest> ) <RparForKindTest> |
| SchemaContextPath | ::= | <SchemaGlobalContextSlash> ( <SchemaContextStepSlash> )* |
| SchemaContextLocation | ::= | ( ( SchemaContextPath <QNameForItemType> ) \| <SchemaGlobalTypeName> ) |
| LocalName | ::= | <QNameForItemType> |
| NodeName | ::= | ( <QNameForItemType> \| <AnyName> ) |
| TypeName | ::= | ( <QNameForItemType> \| <AnyName> ) |
| OccurrenceIndicator | ::= | ( <OccurrenceZeroOrOne> \| <OccurrenceZeroOrMore> \| <OccurrenceOneOrMore> ) |
| | | |
| AtLeastClause | ::= | <At-Least> IntegerLiteral |
| RankMethodClause | ::= | <Rank-By> <Lbrack> RankItem (<Comma> RankItem)* <Rbrack> (<MethodS> StringLiteral)? (<MethodV> StringLiteral)? |
| RankItem | ::= | <Lpar> (Identifier (<Comma> NumericLiteral)? ) <Rpar> |
| RelaxationOrderClause | ::= | <RelaxOrder> RelaxOrderList |
| RelaxOrderList | ::= | <Lpar> RelaxOrderElem <Rpar><br>\| <Lbrace> RelaxOrderElem <Rbrace><br>\| <Lbrack> RelaxOrderElem <Rbrack> |
| | | |
| RelaxOrderElem | ::= | (RelaxOrderItem (<Comma>(RelaxOrderItem \| RelaxOrderList))*) \| (RelaxOrderList (<Comma> RelaxOrderItem)$^{+}$) |
| RelaxOrderItem | ::= | Identifier \| ( “$” VarName “(” SimplePathExpr (“,” SimplePathExpr)* “)”) |
| | | |
| USEClause | ::= | <Lpar> (<Relabel> \| < Edge> \| <Deletion> \| (<HOrderLpar> IntegerLiteral <Rpar>) \|<HOrder>) (<Comma> (<Relabel> \| <Edge> \| <Deletion> \| (<HOrderLpar> IntegerLiteral <Rpar>)\|<HOrder>))*<Rpar> |
| Relabel | ::= | "RELABEL" |
| Edge | ::= | "EDGE" |
| Deletion | ::= | "DELETION" |
| HOrder | ::= | "HORDER" |
| HOrderLpar | ::= | “HORDER(“ |

## References

[1] Shaorong Liu and Wesley W. Chu, Cooperative XML(CoXML) Query Answering at INEX 2003, INEX Workshop 2003

[2] Dongwon Lee "Query Relaxation for XML Model“ In Ph.D Dissertation, University of California, Los Angeles, June 2002

[3] W.W. Chu, H.Yang, K.Chiang, M.Minock, G.Chow, and C.Larson, "CoBase: A Scalable and Extensible Cooperative Information System", Journal of Intelligence Information Systems, 6, 1996.

[4] Dongwon Lee, Murali Mani, Wesley W. Chu "Effective Schema Conversions between XML and Relational Models“ In European Conf. on Artificial Intelligence (ECAI), Knowledge Transformation Workshop (ECAI-OT), Lyon, France, July 2002 (Invited)

[5] http://www.w3schools.com/xml/default.asp

[6] W.W. Chu “Project Description: XML Approximate Matching”

[7] Shaorong Liu, “RLXQuery-EBNF-20.pdf”

[8] Christian Cardenas, “RLXQuery: Parsing and Preprocessing for XML Query Relaxation” UCLA Masters Comprehensive Exam, 2004