Parsing and Preprocessing of RLXQuery for XML Query Relaxation

Christian Cárdenas
Spring 2004

UCLA
Masters Project
Advisor: Professor Wesley W. Chu

1 Abstract

This paper introduces the concept of query relaxation and its early beginnings in the CoBase project, as well as some useful real-world applications of query relaxation. The RLXQuery language expands the power of the CoBase project by applying relaxation to XML databases. The RLXQuery system architecture, which includes a Query Parser, a Preprocessor, a Relaxation Controller, TAH and XTAH Managers, an Execution Trace Manager, and a Configuration Manager, are outlined. The representation of a user's query in RLXQuery is discussed, as there are a number of special object classes used to represent a user's query. An RLXQuery query is not merely a string, but an XQueryRep (XQR) object constructed from a set of XQR class APIs.

My responsibilities included the parser and the preprocessor. In the parser module, I was building upon the previous work of [2]. Significant modifications were necessary in the parser module, particularly with the XRepConvertor, which constructs the XQueryRep object from the output of the parse tree. The other half of my responsibilities lay in the Preprocessor, which translates RLXQuery-specific language constructs to standard XQuery syntax. After working on both of these modules, I constructed a GUI to interact with the integrated parser and preprocessor. At the present time, a user can enter a query in the RLXQuery language and have the query translated through the preprocessor.

2 Introduction

The concept of query relaxation was first introduced with relational databases in the CoBase[1] project at UCLA. In the CoBase project, the SQL query language was extended to CoSQL. CoSQL contains a variety of new language constructs that allow a user to specify fuzzy conditions and that provide relaxation control after a query is executed.

The CoBase project became a successful implementation by the late 1990's. During that time, XML emerged as a competitor to the relational database model. XML quickly became a popular format of information exchange across a number of diversified data sources, such as semi-structured documents, relational databases, and object repositories. With the success and widespread use of XML, there needed to be a way to intelligently use this structured information to perform queries. In 1999, XQuery was developed as a query language applicable across many types of XML data sources. In this current research project, RLXQuery is an extension to XQuery in the same way that CoSQL was an extension to SQL in the CoBase project

XML data sources have a data model that is substantially larger than a relational schema model. Consequently, the user is required to understand far more details in an XML database. With insufficient knowledge of the data structure, a user frequently cannot formulate queries that return any results or even a sufficient number of results. Query relaxation is essential when not enough answers are returned in a user's query. The RLXQuery query language is a solution to this problem. RLXQuery can remove from the user the burden of fully understanding many of

[1] Please refer to http://www.cobase.cs.ucla.edu/ to learn more about CoBase.

the details of the database when performing queries and instead put this responsibility on the query language, which will modify the query appropriately to conform to the user's intentions.

RLXQuery (Relaxation-Enabled XQuery) serves two main purposes. First, it uses a number of additional language constructs, such as approximate values, fuzzy terms, and high-level semantics, to allow a user to formulate queries more easily. The second purpose is to allow a query to return approximate values from a query when exact values are unavailable, which involves relaxing the constraints of the query. Relaxation control constructs, which have also been added to RLXQuery, can be used to facilitate this capability of RLXQuery.

RLXQuery is a superset of the XQuery language. The RLXQuery language includes all of the XQuery constructs, as well new language constructs in addition, which allow for the high-level semantics and provide a wide range of relaxation controls. In order to create the language for RLXQuery, we started with the XQuery EBNF (Extended Backus-Naur Form), which defines the grammar for RLXQuery. This EBNF is provided at the WC3 website[2]. We modified this EBNF by inserting the production rules that allow for the new language constructs, thus defining the RLXQuery grammar.

3 XML Query Relaxation

Query relaxation is the process of understanding the semantic context and intent of a user query and massaging the query constraints into "near" values that provide "best-fit" answers [1]. Query relaxation is a knowledge-based approach to query answering that requires the help of two different knowledge bases, the TAH (Type Abstraction Hierarchy) and the XTAH (XML Type Abstraction Hierarchy). The TAH, which was introduced in the CoSQL language, provides a list of values when approximate values and conceptual terms are used in a query. Additionally, the TAH is consulted for any relaxable value in a constraint when not enough answers are returned from a query. Similar to the TAH, the XTAH is used for approximating path expressions. The XTAH is the component that allows for the breakthrough idea of XML relaxation. Since the hierarchical structure of XML can be quite complex, the XTAH is a valuable knowledge base that recognizes a structurally similar path sequence to the one the user might use in a query. The XTAH can either replace an approximate node in a path expression or it can relax a user's path expression to match a path expression in the XTAH.

4 Applications of Query Relaxation

There are a number of different real-world applications in which query relaxation can be used. One domain is with military aircraft systems. As the example from the CoBase web site, a pilot can ask a conceptual query, "Find a nearby friendly airport where an F-15 can land.[3]" The CoBase system translates the "nearby" conceptual term into a distance range based on the

[2] http://www.w3.org/TR/xquery/

[3] copied from http://www.cobase.cs.ucla.edu/intro.html

position of the aircraft, and the fact that the plane is an F-15 allows the system to know the required runway length and width for landing.

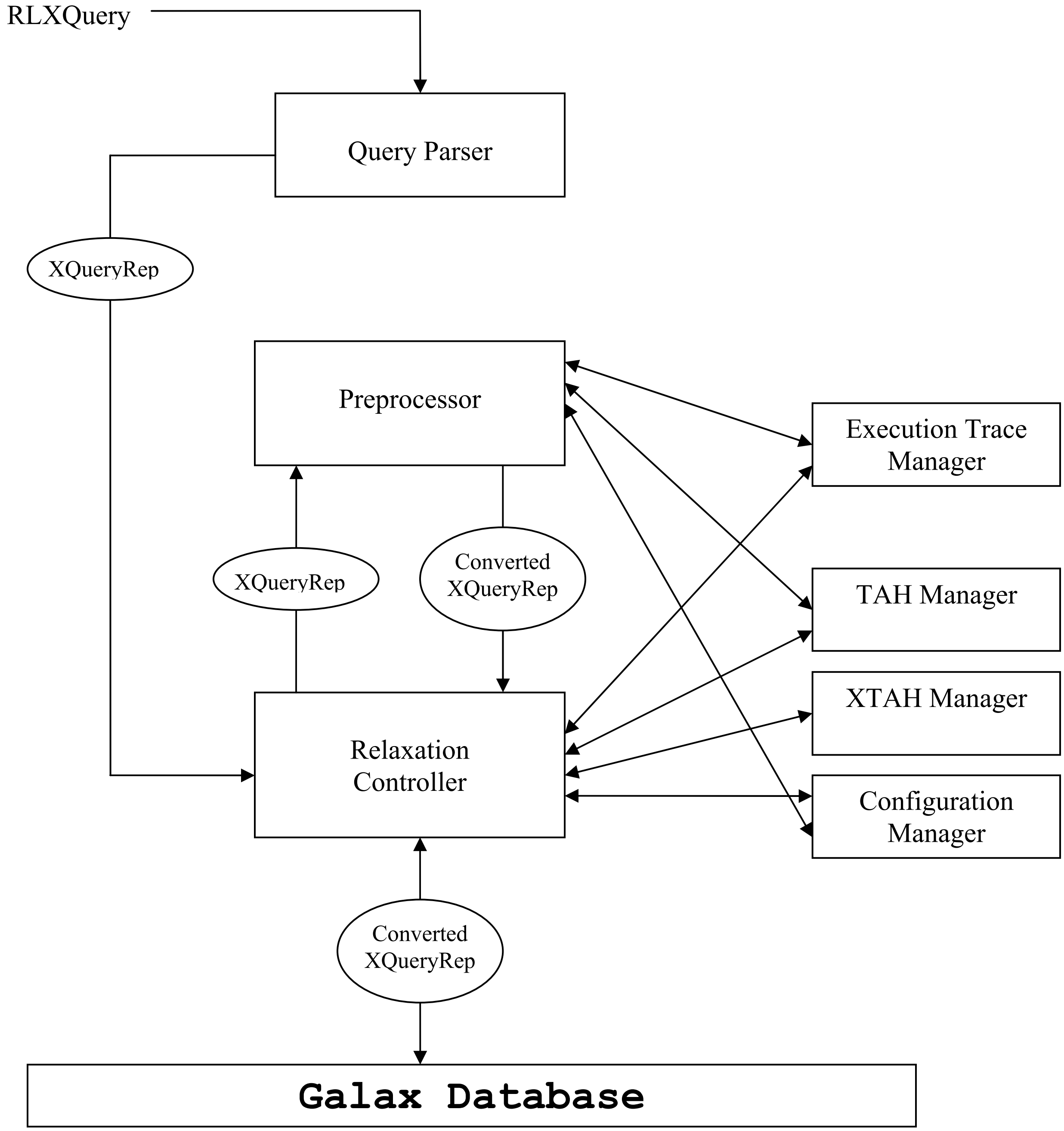
The CoBase project targeted military applications in many of their examples, but there is a wide-range of every-day applications in which query relaxation can be applied. Query relaxation has much potential in online ticket purchasing systems. Many times a user will search for tickets to an event on a particular day in a particular seating section and come up empty. It would be useful for a system to automatically modify his search criteria to find the closest available ticket based on the user's original query. Through the use of a good web interface, a user's preferences can be translated into a query, in terms of which search conditions the user would want to be relaxed automatically. Similarly, the ticketing system could have a user profile in the customer's account, which contains the user's background information and past purchasing history ahead of time. If the user were to search for tickets using high-level semantics, such as finding all tickets on a particular date that are cheap or any tickets that are similar to past purchases, the user's background information can be used to properly translate this search query. The key, however, for effectively using query relaxation in this type of application is for the ticketing system to provide a good user interface that can capture his search criteria in the form of an RLXQuery query.

Another domain in which query relaxation has potential is in car navigation systems. A user, for instance, might want to locate the closest gas stations to him. The navigation system would have to know how to translate the meaning of a gas station to a specific establishment, such as Unocal 76 or Chevron, and then it would have to translate those establishments to a particular set of addresses. The system would also have to know the location of the user in order to know the shortest distances to the set of locations for the gas stations. Furthermore, a smart navigation system might know ahead of time that user would never go anywhere other than to a Chevron or a Unocal 76 station, so it would make these constraints non-relaxable.

5 RLXQuery System Architecture

The diagram below illustrates the different components of the system for the RLXQuery language. There are three main implementation modules: the Query Parser, the Preprocessor, and the Relaxation Controller. There are the peripheral knowledge sources, the TAH and the XTAH. The Configuration Manager contains the user-specified preferences. The Execution Trace manager records all of the preprocessing and relaxation changes to the original RLXQuery query. The last piece of this architecture is the database engine itself. RLXQuery uses the Galax Database. Galax is an open-source implementation of XQuery 1.0 [2].

In this architecture, a user inputs his query. The query is sent to the query parser module where syntax of the query is checked. If the query is valid, the parser will create an XQueryRep object from the user's query. The XQueryRep object is briefly sent to the relaxation controller, where a copy of the XQueryRep object is created and then sent to the preprocessor. The preprocessor checks the XQueryRep object for any special RLXQuery constructs in the query conditions (e.g. approximate operators, conceptual terms, etc.) and translates these constructs to standard XQuery constructs. The preprocessor consults the TAH and XTAH managers when converting approximate and conceptual term operators. It notifies the execution trace manager when it makes any changes to the original query. The converted XQueryRep object goes back to the relaxation controller, which then sends the query to the Galax database where it will be executed. Depending on the results of the query, the relaxation controller will appropriately relax the query with the assistance of the TAH and XTAH managers. The series of relaxations will be recorded by the execution trace manager.



6 RLXQuery Object Representation

Before we can begin to relax a query, there needs to be a way to represent a user's query. Simply representing a query as a String is insufficient. An RLXQuery query needs to store relaxation information in each of the components that make up the query. Each path expression, each condition value, each predicate value, etc. must contain relaxation information. Thus, each component of a query needs to be represented as an object that encapsulates this relaxation information. In order to represent an RLXQuery, there are the XQR (XQueryRep) object classes. Each of these classes begins with the "XQR_" prefix. To represent an RLXQuery query, the topmost object is an XQR_BasicQuery. This object is comprised of many smaller component XQR objects.

7 Components of the RLXQuery System

7.1 Query Parser

Before constructing the query parser, we started out with a grammar file in XML format. This .xml file contains all of the production rules and tokens from the EBNF. The parser encodes the grammar as a set of .java files. Once a query gets sent to the parser, its syntax is checked. If there is an error, then the parser will return and error message and the location of the syntax error. If the parser accepts the query, then a parse tree is returned. The parse tree is a hierarchical display of all the production rules the parser went through when checking the query. The parse tree is the necessary element in creating the XQueryRep object. The parse tree is sent to the XRepConvertor. The XRepConvertor, though not part of the parser itself, is still part of the parser module. The XRepConvertor examines the parse tree and creates the appropriate component XQR objects based on the output in the parse tree. Once the XRepConvertor finishes, the XQR_BasicQuery object is fully constructed and ready for preprocessing.

7.2 Preprocessor

The preprocessor accepts an XQR_BasicQuery object and looks for any special RLXQuery syntax, such as approximate value operators, conceptual terms, SIMILAR-TO constructs, and REJECT constructs[4]. These targeted constructs will be translated to their XQuery equivalent. For approximate values and conceptual terms, the preprocessor needs to contact the TAH Manager in order to find the equivalent exact values. Once these constructs are converted, the query is ready to be evaluated by the relaxation controller. Although there still might be approximate node operators and a number of relaxation operators left in the XQR object, these RLXQuery constructs can simply be omitted when printing the query object as a String.

[4] All of these RLXQuery constructs will be discussed later in more depth.

7.3 Relaxation Controller

The relaxation controller executes the query when it returns from the preprocessor. Depending on the results of the query, the relaxation controller decides whether or not to relax the constraints. If there are no relaxation parameters specified in the query, the relaxation controller will look at all the relaxable (as opposed to explicitly declared non-relaxable) path expressions and values and then make its own decisions on what to relax first. The relaxation controller contacts the TAH and XTAH Managers to figure out how to relax appropriately. If there are explicitly stated relaxation parameters in the query, then the relaxation controller will relax the constraints in the specified order. After each relaxation, the query will be executed. This process continues until a sufficient set of results is returned.

7.4 TAH Manager

The TAH manager contacts a specific TAH (Type Abstraction Hierarchy), which is either specified explicitly in the query or the document on which the query is performed, to find an approximate value for a given value or a set of values for an approximate value. Much of process for creating a TAH can be done automatically. If a TAH is created from an XML document itself, the process of grouping similar values is automatic. For a given path expression in the document, the set of all values can be grouped hierarchically based on their values. This grouping can be done automatically. If a conceptual term is then applied for each cluster in the hierarchy, then this process *would* require human intervention. For example, if the prices of a book were clustered and one of the clusters grouped prices in the range [$0 - $5], then a human can assign the term “cheap” to this cluster. All of this clustered data could be obtained from any element in a path expression such as “$b//books/book//price.”

7.5 XTAH Manager

The XTAH Manager contacts a specific XTAH (XML Type Abstraction Hierarchy), which is either specified explicitly in the query or the document on which the query is performed. An XTAH can find an approximate path expression for the one that needs to be relaxed in the query, or it can find the appropriate node name for an approximate node given in the query. An XTAH is a hierarchical tree-like knowledge structure that contains a summary of path expressions used in the document. Objects in the same cluster are going to be structurally similar. Because of the hierarchical structure of XML data, it is more difficult to cluster nodes and path expressions automatically than it is to cluster values in a TAH.

7.6 Configuration Manager

The Configuration Manager contains a set of user-defined default preferences.

7.7 Execution Trace Manager

The Execution Trace Manager will keep track of every change made to the original user query. Any converted syntax performed in the preprocessor will be recorded. Any series of relaxing operations performed in the relaxation controller will be recorded. Once a query returns the desired results, there should be a summary showing how the query evolved.

8 My Responsibilities

The original focus of my work was on the preprocessor. When I started my work on the preprocessor, the query parser had for the most part been completed by Eric Sung for his Masters project. The preprocessor was just the next component in the system that needed to be done. After finishing much of the preprocessor, it was time to merge these two modules. At this time, the focus of my work shifted to the parser. There were many incompatibilities and issues right from the start. One problem was that the XQR object classes had changed quite a bit since Eric Sung's work. The APIs were different, and many new XQR classes had been added. Many changes needed to be made to Eric's framework when converting the user's query to the XQueryRep object. Another problem was that the RLXQuery language specifications had been updated in the meantime, and, thus, the EBNF was modified quite a bit. The parser needed to be reconstructed using the updated grammar for RLXQuery. It was not an easy task to successfully redefine the production rules in the grammar file and to reconstruct the parser from the new grammar file.

Much of my work involved figuring out how to use the XQR classes and learning to properly create an XQueryRep object. I needed to understand many components of a query object before I could code the preprocessor. The preprocessor involves getting specific component objects and modifying them. After the preprocessor, I had to make many adjustments to the code that converted the input query to an XQueryRep object. (There were a number of mistakes in Eric's code, so that is why substantial work needed to be done.) In order to properly create an XQueryRep object, thorough knowledge of the XQR classes was necessary. I spent much time learning every XQR class so that I could account for every type of query a user might enter.

9 Parser Module

9.1 Constructing the Parser

The RLXQuery parser was automatically constructed with JavaCC (Java Compiler Compiler). , Given a grammar specification for any language, JavaCC automatically generates a Java program (as a set of Java files) that will parse any input that matches the language at hand. In our case, the .xml file that defined all the production rules in the RLXQuery EBNF was passed to JavaCC, and a parser program was automatically generated. JavaCC provides other nice capabilities related to parser generation, such as tree building. JJTree is a class that is used by JavaCC when constructing the parser. The JJTree class enables the parser to represent the parsed input in the form of a parse tree. With a parse tree, it becomes much easier to convert the query to an XQueryRep object.

Eric Sung did the research on JavaCC and learned how to use this tool. He managed to get JavaCC to read an .xml grammar file and have it construct a parser program. Please read Eric Sung's paper to learn more about the parser [3].

9.2 RLXQuery Grammar File

The grammar for RLXQuery is defined in a file called rlxquery_grammar.xml, which will be readable to JavaCC. The grammar file must be formatted with a specific syntax, so all the production rules in the EBNF need to manually converted to .xml format. This process is a long, tedious task. Eric Sung did the entire manual translation in the first version of the RLXQuery EBNF. I had to learn how the grammar file works when I needed to propagate the changes in the EBNF when I began working on the project. This grammar file contains thousands of lines that encode all the terminals and non-terminals in the EBNF.

In the rlxquery_grammar.xml file, each terminal in the EBNF is encoded as a token and its transition state. For each non-terminal in the EBNF, there is a production rule that is defined in terms of tokens and other production rules. The grammar file is defined by three different sections: token elements, state elements, and production elements. The following is an example of how the grammar works.

Tokens:

Comma	::=	","

A terminal in the RLXQuery EBNF

Tokens define the terminals in the EBNF. In the grammar file, the terminal from the above EBNF as defined as such:

```
<g:token name="Comma" node-type="id">
    <g:string>,</g:string>
</g:token>
```

States:

Each token belongs to a particular transition state. A token's transition state looks as such:

```
<g:state name="OPERATOR">
. . .
<g:transition next-state="DEFAULT">
. . .
<g:tref name="Comma"/>
. . .
</g:transition>
```

```
. . .
</g:state>
```

Productions:

BasedOnPathExprList	::=	(SimplePathExpr (“,” SimplePathExpr)*) | (PathExprWeightElem (“,” PathExprWeightElem)*)

A non-terminal in the RLXQuery EBNF

Each non-terminal in the EBNF is defined by a production element in the grammar file. The above non-terminal, BasedOnPathExprList, looks as such:

```
<g:production name="BasedOnPathExprList" if="xquery core">
        <g:choice>
                <g:sequence>
                        <g:ref name="SimplePathExpr"/>
                        <g:zeroOrMore>
                                <g:ref name="Comma"/>
                                <g:ref name="SimplePathExpr"/>
                        </g:zeroOrMore>
                </g:sequence>
                <g:sequence>
                        <g:ref name="PathExprWeightElem"/>
                        <g:zeroOrMore>
                                <g:ref name="Comma"/>
                                <g:ref name="PathExprWeightElem"/>
                        </g:zeroOrMore>
                </g:sequence>
        </g:choice>
</g:production>
```

In the .xml grammar file, a “|” from the EBNF is represented with the <g:choice> … </g:choice> tags. Similarly, a “*” is represented by the <g:zeroOrMore> … </g:zeroOrMore> enclosing a sequence. The <g:sequence> … </g:sequence> tags group a sequence of elements together as one entity within a <g:choice> tag. Creating the grammar file required converting every production rule from in the ENBF in a similar manner.

9.3 JJTree

JJTree is a JavaCC's object representation of its parse-tree. The main function of JJTree is to build an independent parse-tree representation of the query that is being parsed. Every node in the parse tree is either a terminal or a non-terminal. Terminal nodes have a corresponding data value. When JJTree creates the parse tree, it gives the top node of the parse tree. Each node is a SimpleNode object. The SimpleNode class has a number of tree traversal methods, such as jjtGetNumChildren( ), jjtGetChild(int), and getSubTree(String). These methods make it possible to traverse downward through the tree look for specific nodes. It is during this tree traversal that the XQueryRep object is created. Based on the number of children and any subtrees of a node, the XQueryRepConvertor knows what XQR component objects needs to be created for the query. For example, at a certain node in the parse tree that signifies the start of a path expression, one could make a call to get the subtree for the "SpecialPredicates" node and then make another call to get the number of children. If zero is returned, it means there are no predicate conditions on particular path expression. This same approach is taken to convert the entire the query to object form. Please refer to the appendix to see a sample parse tree.

10 XQueryRep Object

This is an in-depth look at the key component objects that make up an XQueryRep object and how to construct them properly. The following example below illustrates the object representation of an RLXQuery.

10.1 Sample RLXQuery

By showing an example, the method of creating an XQueryRep object should become much clearer. Start with the following RLXQuery:

```
for $b in document("books.xml")//books/book
where ($b/author/first = "Christian" and $b/author/last = "Cardenas") and
        $b/year = ~2000 USE-TAH ApproxPublicationsTAH
return $b/title
```

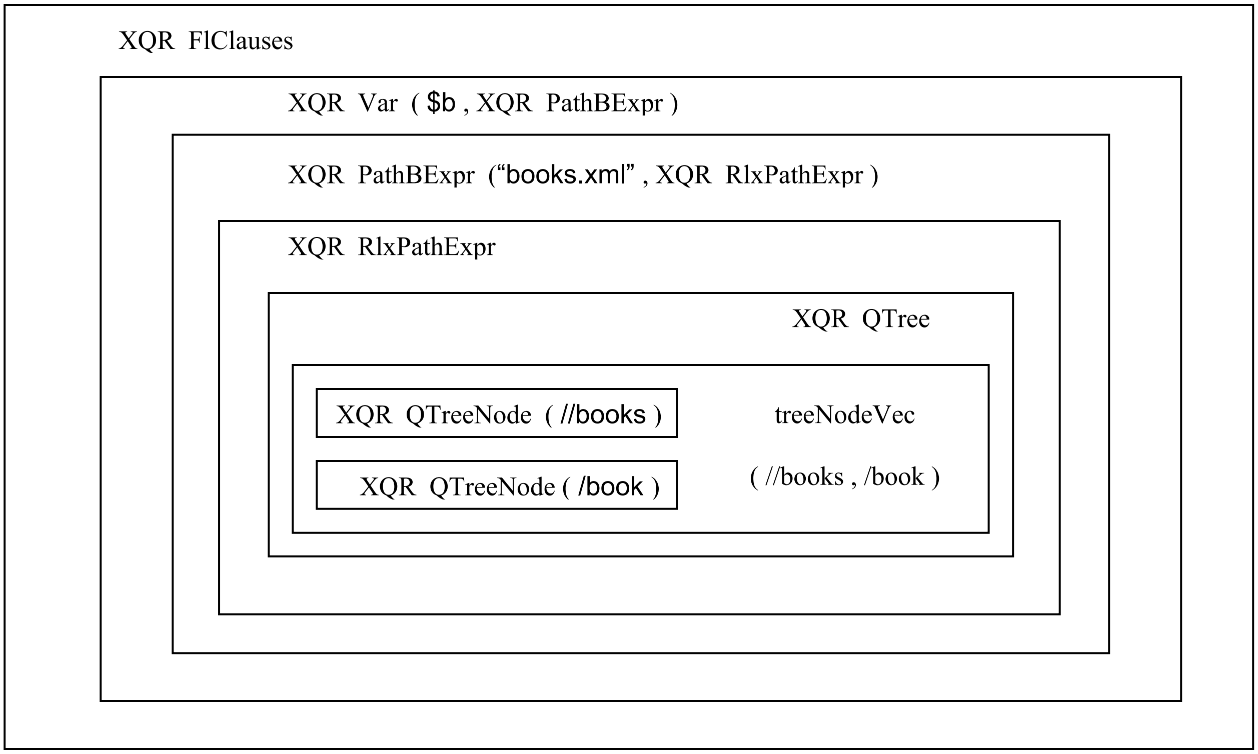
The top-most class is the XQR_BasicQuery class. At the time of instantiation, this class requires for/let clauses and a return expression. The return expression is simply a string. On the other hand, the for/let clauses are not simply strings, but instead another XQR class in itself. A where condition can optionally be set later with a set_where_cond(XQR_whereCond) method call.

The for/let clauses are represented by the XQR_FLClauses class. In this class, there is a hashtable that maps each variable name to its corresponding XQR_Var object. The XQR_Var object stores the string representation of the variable name, a flag that indicates for or let expression, and the binding path expression as an XQR_PathBExpr object. In this example, the XQR_PathBExpr object contains a document name and an XQR_RlxPathExpr object. The XQR_RlxPathExpr object simply contains an XQR_QTree object. The path expression itself is stored as an XQR_QTree, which will be discussed separately in a further section. Creating a path expression as a tree – which contains a sequence of XQR_QTreeNodes – is a large enough task by itself.

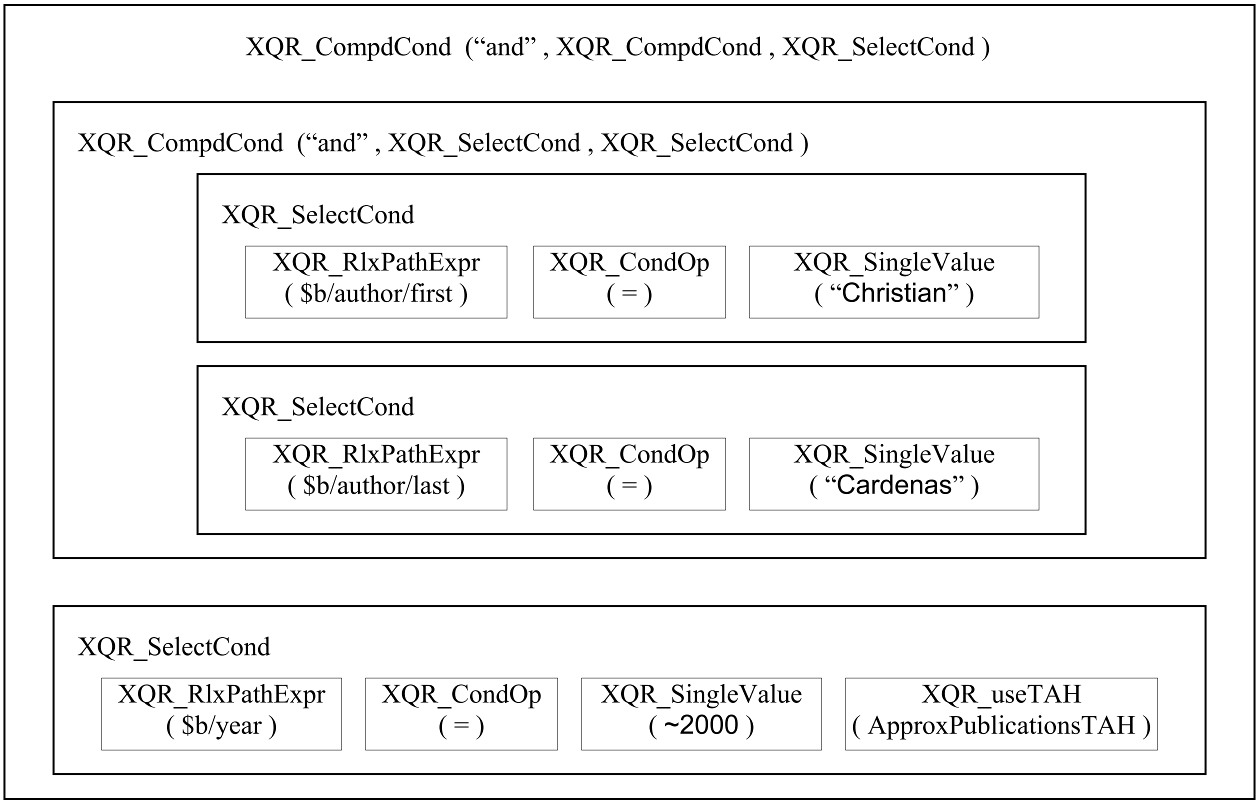
Once the for/let clauses and the return expression is set, the where condition needs to be set (if there are any). The where condition class, XQR_whereCond, can be instantiated with an XQR_selectCond object or with an XQR_CompdCond object. In the example below, the where condition contains an XQR_CompdCond object that has an XQR_CompdCond object as its left child and an XQR_SelectCond object as its right child. In this example, each XQR_SelectCond contains a path expression (XQR_RlxPathExpr object), a condition operator (XQR_CondOp object), and a value (XQR_SingleValue object). In order to create more than two conditions in

the where clause, recursively create XQR_CompdCond objects as the children of an XQR_CompdCond object, just as in the example below. In the last XQR_SelectCond object, there is a USE-TAH expression. This is done simply by calling the set_use_tah(XQR_useTah) method. The XQR_useTah class is a small class that contains the name of the TAH.
The diagram below shows what the above sample query looks like as an XQueryRep (XQR) object.

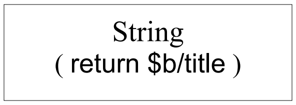
for $b in document("books.xml")//books/book


XQR FlClauses
XQR Var ($b , XQR PathBExpr)
XQR PathBExpr ("books.xml" , XQR RlxPathExpr)
XQR RlxPathExpr
XQR QTree
XQR QTreeNode (//books)
XQR QTreeNode (/book)
treeNodeVec
(//books , /book)

**where ($b/author/first = "Christian" and $b/author/last = "Cardenas") and
$b/year = ~2000 USE-TAH ApproxPublicationsTAH**


XQR_CompdCond (“and” , XQR_CompdCond , XQR_SelectCond)
XQR_CompdCond (“and” , XQR_SelectCond , XQR_SelectCond)
XQR_SelectCond
XQR_RlxPathExpr ($b/author/first)
XQR_CondOp (=)
XQR_SingleValue (“Christian”)
XQR_SelectCond
XQR_RlxPathExpr ($b/author/last)
XQR_CondOp (=)
XQR_SingleValue (“Cardenas”)
XQR_SelectCond
XQR_RlxPathExpr ($b/year)
XQR_CondOp (=)
XQR_SingleValue (~2000)
XQR_useTAH (ApproxPublicationsTAH)

return $b/title



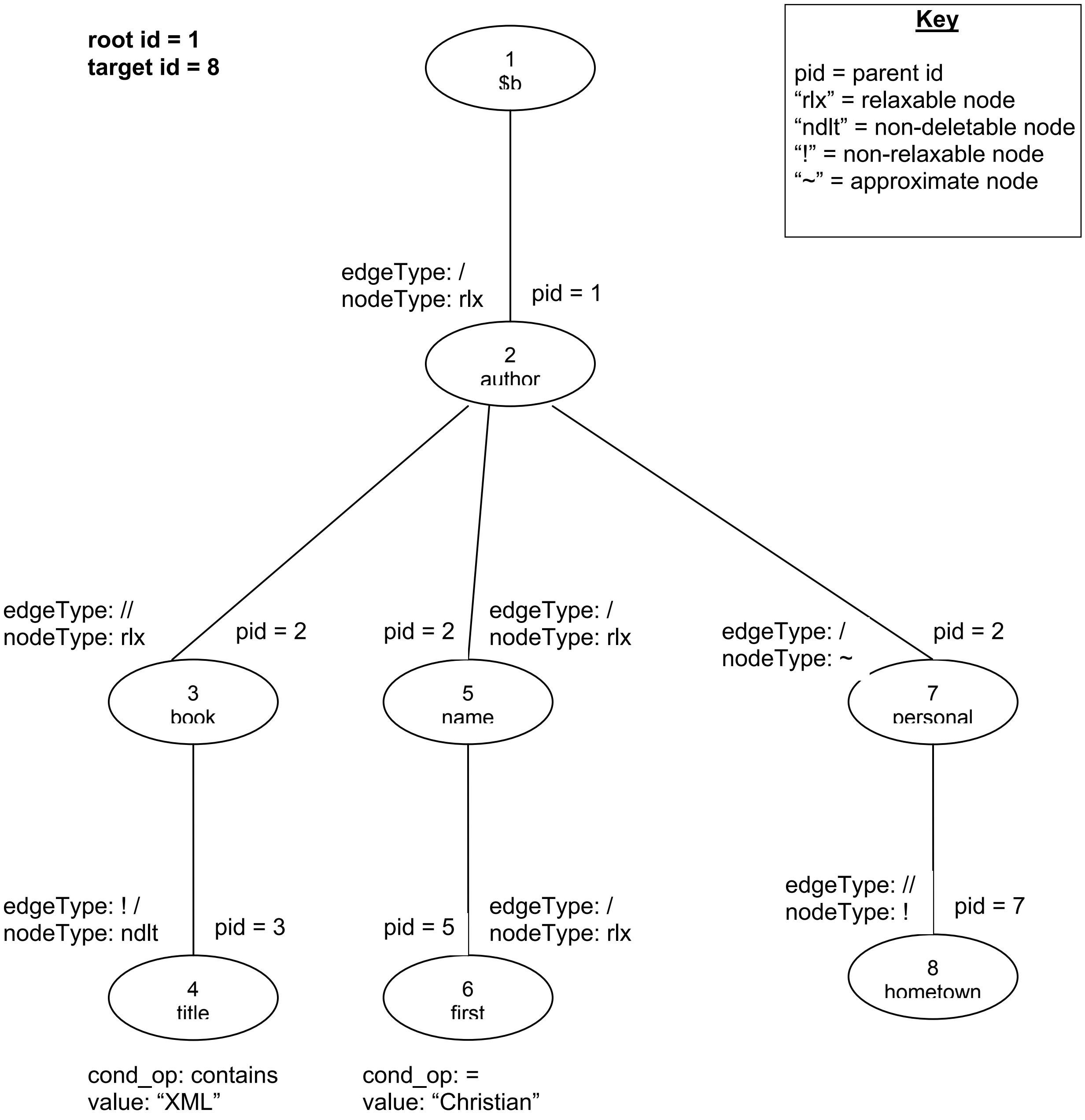
10.2 Path Expressions as QTree Objects

A path expression, assuming it is instantiated as an XQR_RlxPathExpr object, is logically represented as a tree of XQR_QTreeNode objects. These XQR_QTreeNode objects are stored in a treeNodeVec, which the XQR_QTree class stores. In turn, the XQR_RlxPathExpr class contains an XQR_QTree object. Each node contains a node name, node relaxation information, edge information, and a parent node identification number. Additionally, a node might have a predicate value condition. Once all the nodes are created and stored in a treeNodeVec object, the XQR_QTree object stores the root node identification and the target node identification.

The best way to understand the process of creating a path expression is by an example. The following example is a relatively complex example, but it fully demonstrates the capabilities of the XQR_QTreeNode class.

$b/author[.//book!/title[contains(., “XML”)] and name/first = “Christian”]/~personal//!hometown

There are two predicate conditions, one on the node title and another on the node first. The predicate conditions make the XQR_QTree a bit tricky to create automatically from the parse tree. Some of the nodes and edges also contain special relaxation information, which must be taken into account when creating each XQR_QTreeNode. This path expression looks like the following as an XQR_QTree object:



When actually creating the XQR_QTreeNode object, the edge type is stored with a 0, 1, 2, and 3 for “/”, “//”, “! /”, and “! //” edges, respectively. The node type is stored with a 0, 1, 2, and 3 for a normal (relaxable) node, a non-relaxable node, a non-deletable node (a node with a non-relaxable edge type), and an approximate node, respectively. Please refer to the documentation for the XQR_QTreeNode class to read more.

In the example, the first node in the path expression, $b, is a variable. This node is created a bit differently than the others. This node has already been created and bound to a path expression in the for/let clauses. Thus, the XQR_QTreeNode for $b is created by passing the XQR_Var object for the variable $b, with node identification as 0. In order to get the corresponding XQR_Var object for $b, one can get the hashtable for the XQR_FLClauses object and then get the corresponding XQR_Var object for the key "$b."

There are a few other remarks to make about the above example. 1) The root node is first node in the path expression. The target node is the last node in the expression that is *not* in a predicate. In this case, hometown is the target node. 2) Predicates are any nodes that are within "[…]." Thus, book, title, name, and first are all predicate nodes. A predicate node can have a value condition. The condition operator and the value are set through method calls after the XQR_QTreeNode object has been instantiated. In the example above, the contains operator is considered a condition operator, with "XML" being the condition value, in the XQR_QTreeNode object. 3) Any predicate node not prefixed with anything is assumed to have same meaning as a node prefixed with "./" Thus, in the example, the predicate node name/first also could have been written as "./name/first"

There are some path expressions that cannot be represented with a series XQR_QTreeNodes. Any path expression that has multiple conditions on one particular predicate node *or* any path expression that has two predicate conditions joined by an "or" operator are treated as a complex path expression. Instead an XQR_ComplxPathExpr object, which represents a path expression as a string, is used. Here is an example of each case:

Case 1: $b/title[contains(., "XML") and contains(., "HTML")]

Here, there are two predicate conditions on the node title. A XQR_QTreeNode can have only one predicate condition and value.

Case 2: $b/author[name/last = "Cardenas" or name/first = "Christian"]!//birth

Here, two predicate conditions (which are each for a different node) are joined by and "or" operator. In the XQR_QTree structure, there is no way to represent the "or" condition. Two predicate subtrees are assumed to be related by an "and" condition. Thus, such a path expression needs to be represented as a string.

As a general remark, the more types of path expressions that can be represented as a tree of XQR_QTreeNodes, the better. The node structure allows the relaxation controller to better understand the structure of a path expression during the relaxation process. The relaxation controller needs to know information about each node and their relationship to other nodes when determining with node or which part of the path expression to relax (i.e. rewrite for the purposes of executing the query). Representing a path expression as a string does not allow the relaxation controller to use its relaxation abilities, which include node re-labeling, node-deletion, edge relaxation, and order relaxation. These four types of path relaxations will be discussed in a further ahead.

11 Preprocessor

11.1 Targeted Language Constructs

The preprocessor looks for certain language constructs that can be re-written in standard XQuery form. There are four types of constructs the preprocessor translates to standard XQuery.

1. Approximate value operators: These are values prefixed with a "~". The TAH Manager is consulted in order to find an equivalent range or an equivalent set of values to substitute for the approximate value. In the where condition, a specific TAH to use can optionally be declared. If no TAH is specified, then the document on which the query is performed becomes the TAH of choice. An example using the approximate operator is the following:

   ```
   for $b in document("books.xml")//books/book
   where $b/price = ~20
   return $b
   ```

   If the TAH Manager, which uses "books.xml" as the TAH, along with the path expression $b/price, returns ([18, 20], 22, 24) as the approximate values in the given context, then the preprocessed query is:

   ```
   for $b in document("books.xml")//books/book
   where $b/price = ( [18, 20], 22, 24 )
   return $b
   ```

   Note that [18, 20] denotes a range value. Range values are legal values in standard XQuery, and this range is written as (18 to 20). The as_string( ) method for each XQR_whereCond object knows to print range values is this XQuery format.

2. Conceptual terms: These values are prefixed with a "#" symbol. The way conceptual terms are handled is very similar to the way approximate values are handled. A TAH Manager is also contacted in order to find an equivalent set of values that define the conceptual term in the given context. An example is the following:

   ```
   for $b in document("airports.xml")//airports
   where $b/location = #"near" USE-TAH AirportsTAH
   return $b
   ```

   The TAH "AirportsTAH" is used to convert the conceptual term "near." Based on the context of the user query, the TAH might return the range [0, 200] as the range of distances that it considers an airport's location "near" in the given context. This range value is substituted for "near".

3. Reject clauses: A reject clause uses the "REJECT" construct, which enforces that that particular condition can never be relaxed to use to values given in the reject clause. A reject clause can be used with a where condition that uses a PREFER construct or with a where condition that has an approximate value operator. A reject clause can also be used in a where condition that does not use any other RLXQuery constructs. Some examples are the following:

   ```
   where $b/name = PREFER("Christian", "Shaorong") REJECT("Victor", "Laura")

   where $b/year = ~2000 REJECT(1995, 1998, [2002, 2004])

   where $b/price = 2.50 REJECT(2.00, [2.75, 3.00], 3.50)
   ```

   In each case, a reject clause is simply translated to standard XQuery syntax by adding another condition in the where clause that makes sure the reject values are not equal to the path expression *and* making each value in the value list *non-relaxable*. Thus, the above where condition will look as such after preprocessing:

   ```
   where $b/name = PREFER("Christian", "Shaorong") and
           $b/name != (!"Victor", !"Laura")

   where $b/year = ([1998, 2000], 2001, 2003) and
           $b/year != (!1995, !1998, ![2002, 2004])

   where $b/price = 2.50 and
           $b/price != (!2.00, ![2.75, 3.00], !3.50)
   ```

   In the second example, assume the TAH Manager returned ([1998, 2000], 2001, 2003) for the approximate value 2000. We see that there is overlap between the TAH's values and the reject values (e.g. 2003). Even though there is overlap, the reject value still will never be used in the where condition since the reject values are non-relaxable.

4. Similar-To clauses: These clauses use the "SIMILAR-TO" construct followed by a "BASED-ON" construct. The following query is an example:

   ```
   for $c in document("cdstore.xml")//cdstore
   where $c/name SIMILAR-TO "Hollywood" BASED-ON ($c/address/city, $c/rating)
   return <cdstore> {$c/name, $c/rating} </cdstore>
   ```

   What this query means is, "Find the name and rating of any store that is in the same city and has the same rating as the store named "Hollywood." There are two steps involved in translating this query. The first step in the translation of the query requires evaluating a query in Galax database. (This is the only situation in the preprocessor that involves using the database to execute a query.) In the first step, the query is re-written as the following and then executed in Galax:

```
for $c in document("cdstore.xml")//cdstore
where $c/name = "Hollywood"
return <tmp> { $c/address/city, $c/rating } </tmp>
```

Suppose the database engine returns the tuple ("Los Angeles", 5.0). The second step requires creating the where condition from the results of this intermediate query and using the original return condition in the preprocessed query. The preprocessed query looks as such:

```
for $c in document("cdstore.xml")//cdstore
where $c/address/city = "Los Angeles" and $c/rating = 5.0
return <cdstore> {$c/name, $c/rating} </cdstore>
```

The above example illustrated a single SIMILAR-TO construct. Multiple values can be used after the SIMILAR-TO clause as well. For example, consider the following query:

```
for $c in document("cdstore.xml")//cdstore
where ($c/name, $c//address/city) SIMILAR-TO ("Hollywood", "Los Angeles")
        BASED-ON (($c/rating, 0.6), ($c//video_categories, 0.4))
return <cdstore> {$c/name, $c/rating} </cdstore>
```

This query means, "Find the name and rating of any store that has the same rating and the same video categories as the store with the name "Hollywood" and the location Los Angeles. This query is converted the same way the single SIMILAR-TO example is, except that in the intermediate form of this query, there are two conditions in the where clause, which looks as such:

```
where $c/name = "Hollywood" and $c//address/city = "Los Angeles"
```

Other than this, the multiple SIMILAR-TO situation is just like the single SIMILAR-TO. Note that there are weights associated with each path in the BASED-ON clause. These weights pertain to the relaxation controller should the query need to be relaxed.

These four different targeted language constructs are not be confused with the relaxation constructs, which are not able to be translated. These four constructs are like high-level language constructs, as opposed to relaxation controls. (The REJECT clause is considered a relaxation control, but it can be translated to standard XQuery form so that the relaxation controller does not have to deal with it.) The relaxation controls can easily be masked out when the XQR object prints itself right before sending it to the database engine. Relaxation controls (which will be discussed in a further section) are constructs that are appended to the end of the query or at the end of a condition in a where clause. The as_string( ) method takes a flag that determines whether or not to print the XQR object in XQuery form or RLXQuery form. If the preprocessor has not translated any approximate values, conceptual terms, reject clauses, or SIMILAR-TO constructs, then the XQR object cannot possibly print itself in standard XQuery form.

The one exception is the PREFER clause, which is a construct that cannot be masked out optionally in the as_string( ) method. There was debate whether or not the preprocessor should translate the PREFER clause (by using the first value in the value list as the value in the condition). After some thought, it was decided to have the relaxation controller convert the PREFER clause since a PREFER clause function just like a relaxtion control, which contains a list of values to use in a condition in a specified order.

One more thing to note is the use of the range value in RLXQuery, which uses square brackets. A range, such as [0, 10] in RLXQuery, is not valid syntactically in XQuery. In XQuery, this range would be written as "(0 to 10)". The as_string( ) method on the XQR object knows to print out a range value in standard XQuery form. In XQuery, ranges must have integer values only, but in RLXQuery, ranges are permitted to have double values as well. To work around this XQuey limitation, the as_string( ) method for the XQR object check if the range is a double range. If so, then it will specially print the range value as a conjunction of inequalities. As an example, the RLXQuery range expression, "$b/value = [0.01, 9.99]", is printed as "$b/value >= 0.01 and $b/value <= 9.99" in the as_string( ) method.

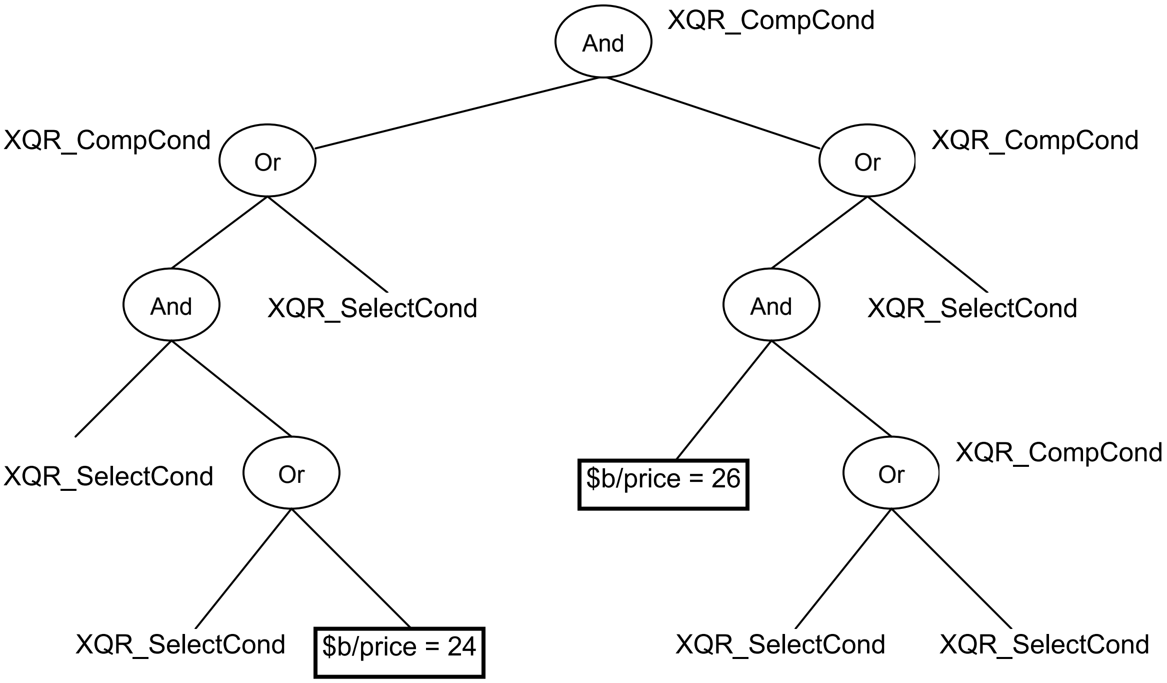
11.2 Algorithmic Design of Preprocessor

The preprocessor examines only the where clause in a query. When a query is sent to the preprocessor, the first thing is to check if the where condition is of type XQR_CompdCond. If so, then two recursive calls are made, which examine the left child of the compound condition and then the right child of the compound condition. Once the where condition is of type XQR_SelectCond, the where condition can be examined for the targeted RLXQuery constructs. First, if a reject clause exists, it will be translated. Thus, an XQR_CompdCond will be created, replacing the old XQR_SelectCond. Second, any approximate values or conceptual terms will be substituted with the list of values returned from the TAH Manager. Lastly, if there is a SIMILAR-TO construct, it will be translated with that two-step process.

11.3 Other Functions of Preprocessor

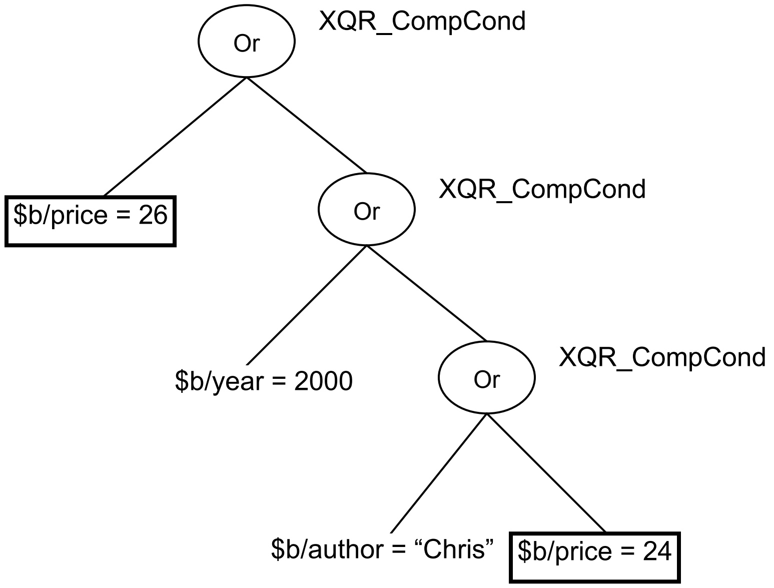
Another feature of the preprocessor that is in progress is the process of merging two conditions in the where clause that have identical path expressions. The idea is to make the query execute more efficiently with one less condition to evaluate. For example if there are two path expressions, such as "$b/price = 24 or $b/price = 26", then the preprocessor can recognize that these two conditions can be written as "$b/price = (24, 26)".

There are a number of difficulties involved with implementing such a feature. One difficulty is the fact that two conditions with identical path expressions might located in different parts of the tree of the root XQR_CompdCond object. For example, an XQR_CompdCond object might have a structure that looks as such:



In the above example the path $b/price occurs twice and can be merged. However, starting from the top and getting to $b/price = 24 requires a recursive tree traversal. Finding an identical path expression in the subtree on the other side of the tree requires traversing back to the top and down the right child subtree. Such an algorithm would be tough to implement. Once this step is even accomplished, there still needs to be a way to check if merging two conditions with the same path expression will change the overall meaning of the where condition. This would be very difficult to do.

In order to simplify the idea of merging matching path expressions, for now the preprocessor will be limited to looking at path expressions that 1) have a descendant/ancestor relationship in the tree structure and 2) are connected by the same types of internal nodes (i.e. all "Or" nodes or all "And" nodes). Such a relationship is illustrated by the following tree:

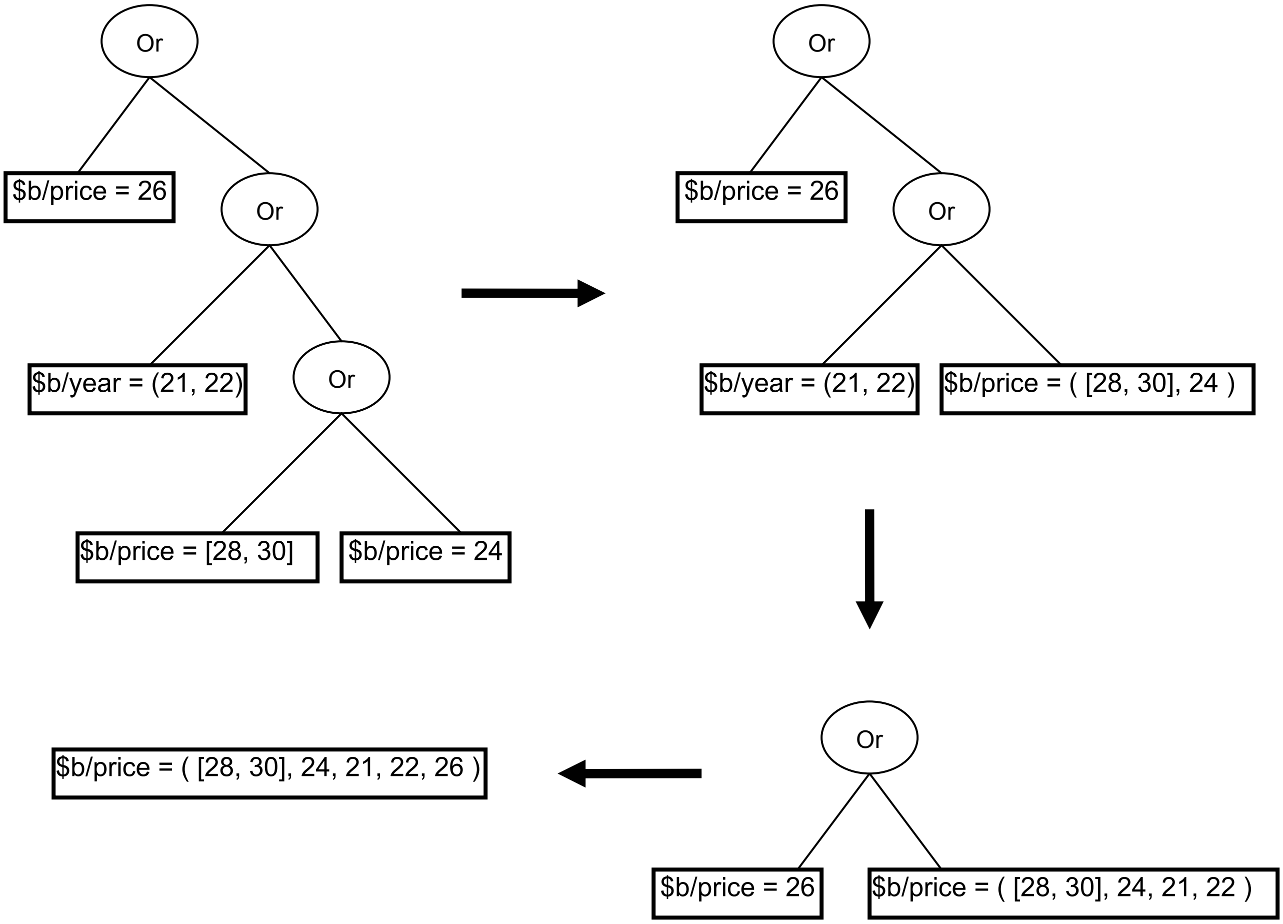


This tree represents the path expression "($b/price = 26 or ($b/year = 2000 or ($b/author = "Chris" or $b/price = 24)))", which can easily be converted to "($b/price = (26, 24) or ($b/year = 2000 or $b/author = "Chris"))" without changing the meaning of the original where condition. In this case, the parent of "$b/price = 26" is an ancestor of "$b/price = 24", and every internal node between those two leaves is of type "Or." If the parent of "$b/price = 26" were not an ancestor of "$b/price = 24", it would not be easy to merge identical path expressions, so we do not examine two nodes with such a relationship.

If two ancestor/descendant nodes are connected by all "Or" nodes, then the two right hand operators can be merged into a list of values (as in the example above). This assumes that the value operator for each expression is a "=". A list of values has the same meaning as a series of "Or" expressions. If two ancestor/descendent nodes are connected by all "And" nodes, then the value operators on each expression must be inequality operators, in order to merge the two expressions. The following shows the simple cases that can be merged easily:

Expression 1	Expression 2	Logic Op	Merged Expression
$b/price = 24	$b/price = 26	or	$b/price = (24, 26)
$b/price >= 24	$b/price =< 26	and	$b/price = [24, 26]
$b/price < 24	$b/price > 26	or	$b/price != [24, 26]
$b/price < 24	$b/price < 20	and	$b/price < 20
$b/price < 24	$b/price < 20	or	$b/price < 24

If more than two expressions can be merged, the preprocessor will sequentially merge a pair of expressions at a time. For example the expression tree will be merged in the following manner:



When sequentially merging expressions, even if each expression has the same path expression, the merged result must make sense. The pair of expressions must classify as one of the simple cases listed in the table above. For example, if "$b/price = 24" and "$/price = [30, 32]" were the pair of expressions, there is no way to merge these two conditions.

12 RLXQuery Language

This section lists and shows examples of every special RLXQuery construct. This section was adapted from Shaorong Liu's "XML Query Relaxation Constructs" document [5].

12.1 High Level Constructs

These are all the constructs that the preprocessor will detect and translate. The REJECT clause is considered a relaxation control, but the preprocessor handles this control. The implementation of the RLXQuery engine treats it as though it were a high level construct that can be translated to standard XQuery form.

Approximate operator (~)

The approximate value operator can be used with any numerical or non-numerical constant value or expressions in a WHERE clause.

Example 1: (Approximate Value Operator, approximate numerical value)

```
for $b in document (“bib.xml”)//book
where $b/publisher = ‘Morgan Kaufman’ and $b/year = ~2000
return $b/title
```

Example 2: (Approximate Value Operator, approximate non-numerical value)

```
for $b in document (“cars.xml”)//car
where $c/make = ‘Ford’ and $c/color = ~”RED”
return $c
```

Conceptual Terms (#“…”)

This is used for values only. Conceptual terms are used as normal values as long as the terms are defined as valid TAH nodes. A conceptual term is prefixed with a “#” and can be used only in a WHERE clauses. They cannot be used in a predicate in a FOR/LET variable binding expression.

Example 3:

```
for $b in document (“bib.xml”)//book
where contains ($b/title, “XML”) and $b/price = #“cheap”
return <book>{$b/year, $b/title}</book>
```

SIMILAR-TO construct

There are two forms of the SIMILAR-TO construct. The syntax of the first form is:

PathExpr$_1$ SIMILAR-TO value$_1$ BASED-ON ((PathExpr$_2$, [weight$_2$]), ... , (PathExpr$_N$, [weight$_N$]))

The syntax for the second form of SIMILAR-TO is:

PathExprList SIMILAR-TO ValueList BASED-ON ((PathExpr$_2$, [weight$_2$]), ... , (PathExpr$_N$, [weight$_N$]))

By default, all weights are 1. The PathExprs and weights are used to match the MTAH (or TAHs) to be used, or an MTAH may be generated on the fly based on these conditions. If there is no BASED-ON clause specified, the system will look up default criterion for the user and find an appropriate TAH to use. Similar-to operator finds answers that are “similar to” or “near to” the object BASED ON some criteria. These are typically used where normal operators are used in WHERE clause of FLWOR expressions. SIMILAR-TO (BASED-ON) cannot be used in predicates.

Example 4: (Single SIMILAR-TO)

```
for $c in document (“cdstore.xml”)//cdstore
where $c/name SIMILAR-TO “Hollywood” BASED-ON ($c/address, $c/rating)
```

```
return <cdstore> {$c/name, $c/rating}</cdstore>
```

Example 5: (Multiple SIMILAR-TO)

```
for $c in document (“cdstore.xml”)//cdstore
where ($c/name, $c//address/city) SIMILAR-TO (“Hollywood”, “Los Angeles”) BASED-ON
(($c/rating, 0.6), ($c//video_categories, 0.4))
return <cdstore> {$c/name, $c/rating}</cdstore>
```

REJECT construct

The REJECT operator is used to specify a set of non-acceptable value relaxation. The syntax for the REJECT operator is as follows:

PathExpr$_1$ Operator value$_1$ REJECT(value$_2$, value$_3$, .., value$_N$)

where value$_i$ can be a string literal, numeric literal, or numerical range.

Example 6: (REJECT construct following a relaxable value)

This query finds a book on XML and published in 2003. If there is no exact match, the book's publishing year is relaxed, but books published in 1998 or before are not acceptable.

```
for $b in document (“bib.xml”)//book
where contains ($b/title, “XML”) and $b/year = 2003 REJECT([0, 1998])
return <book> {$b/title, $b/year} </book>
```

Example 7: (REJECT construct following a PREFER construct)

```
for $b in document (“bib.xml”)//book
where $b!/~title[contains(., “XML”)] and
count($b//section) = PREFER(5, [10, 15]) REJECT ([0, 4], 20 )
return $b
```

12.2 Relaxation Constructs

There are two categories of relaxation constructs: local relaxation control operators and global relaxation control operators. Local operators are found in there where condition, and they apply to the particular XQR_SelectCond object at hand. Global operators are found at the end of the query after the return statement. They apply to multiple XQR_SelectCond objects at a time or to the entire query as a whole.

12.2.1 Local Relaxation Control Operators

Non-Relaxable Operator (!)

The non-relaxable operator is used to specify the set of conditions which should not be relaxed during the query relaxation process. It can be used to express non-relaxable value conditions, non-relaxable path expressions, and non-relaxable nodes and edges in a path expression.

Non-relaxable value

The syntax for a non-relaxable value is:

!(ExactValueExpr)

where ExactValueExpr represents a string, a numerical literal, a numerical range, or a numeric list. For example, the following RLXQuery finds a book published in the year 2003 (which is non-relaxable) with title containing "XML".

Example 7:

```
for $b in document ("bib.xml")//book
where $b[contains(./title, "XML")]/year = ! 2003
return <book> {$b/title, $b/price} </book>
```

Non-relaxable structure

The syntax for a non-relaxable structure is represented as:

!(SimplePathExpr)

where SimplePathExpr represents normal XQuery path expression without predicates. For example, the following RLXQuery finds a book published in year 2003 and with title containing "XML". The structure condition of //book/year is non-relaxable. It returns the book's title and price information back to user.

Example 8:

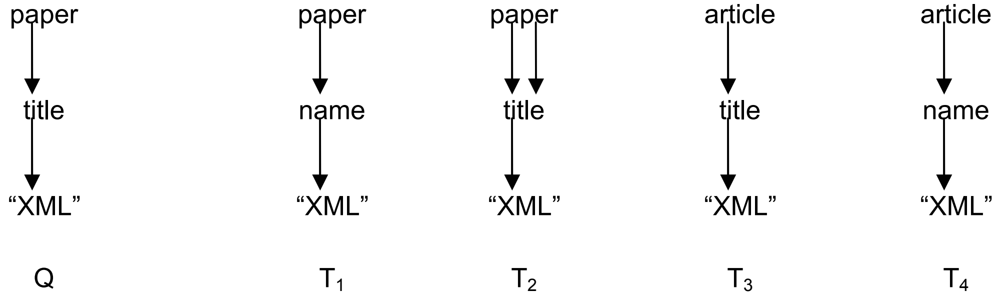
```
for $b in document ("bib.xml")//book
where !($b/year) = 2003 and $b/~title[contains(., "XML")]
return <book> {$b/title, $b/price}</book>
```

Non-relaxable node in the structure

Non-relaxable node means this node cannot be deleted and it cannot be relabeled. For example, "/!paper/title[contains(., 'XML')]" means that node paper is non-relaxable. In the diagram below, this path expression can be relaxed to T_1 and T_2, but *not* to T_3 and T_4.

Non-relaxable edge in the structure

The syntax is: ! Edge, where an edge is either a parent-to-child edge, i.e., "/", or a ancestor-to-descendant edge, i.e., "//". For example, "/paper!/title[contains(., 'XML')] "means the parent-to-child relationship between node paper and node title are non-relaxable. In the diagram below, this path expression can be relaxed to T_1, T_3, or T_4, but *not* to T_2.



Example 9: (Use of a non-relaxable node and non-relaxable edge)

```
for $b in document ("bib.xml")//!book
where $b !/title [contains(., "XML")] and $b//~year > ! 2000
return $b
```

PREFER Construct

The PREFER operator is used to specify some values as preference if an exact matched value is not available. The syntax for the PREFER operator is: path expression = PREFER ($value_1$, $value_2$, … , $value_n$). Note that RLXQuery currently only supports a PREFER construct with an **equal** operator. The PREFER construct **cannot** be applied to path expression in a predicate condition, whether in a where clause or a FOR/LET variable binding expression.

Example 10:

```
for $b in document ("bib.xml")//book
where $b/publisher = PREFER ("MIT", "AW", "PHI") and $b/year > 2000
return <book>{$b/publisher, $b/year}</book>
```

Another use of the PREFER construct is the following:

count(SimplePathExpr) = PREFER($value_1$, $value_2$, … $value_n$)

Example 11: (Use of a PREFER construct with the count where condition)

```
for $b in document ("bib.xml")//book
where $b/publisher = PREFER ("MIT", "AW", "PHI") and count($b//figure) = PREFER (10, 15)
return <book>{$b/publisher, $b/year}</book>
```

USE-XTAH/USE-TAH

This operator has the following usage: 1) to specify that a particular XTAH (or TAH) to be used for the query relaxation process; 2) to label the condition for relaxation order control in RELAX-ORDER constraints; and 3) to combine the first & second;

Example 12: (Use of USE-XTAH and USE-TAH in the same query)

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] USE-XTAH t1 and ($p/year > 2000) USE-TAH t2
return <answer> {$p/title, $p/year}</answer>
```

S-COND-ALIAS / V-COND-ALIAS

The V-COND-ALIAS (S-COND-ALIAS) operator can be used to tag the value (structure) part of a condition, respectively, by giving an alias to the value (structure) part of a condition. (Note: In CoBase we only need one COND-ALIAS operator to un-ambiguously tag query condition(s) because in the relational model, we only allow value relaxation.) In the XML model, we allow both structure and value relaxation. Thus, we need two label operations, S-COND-ALIAS and V-COND-ALIAS, to unambiguously associate a tag with the structure or value part of a condition.

Example 13:

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] S-COND-ALIAS c1 and
        $p/year > 2000 V-COND-ALIAS c2
return <answer> {$p/title, $p/year}</answer>
```

V-RELAX-LEVEL / S-RELAX-LEVEL

The V-RELAX-LEVEL (S-RELAX-LEVEL) is used to specify the value (structure) relaxation level for single or conjunctive conditions. V-RELAX-LEVEL (S-RELAX-LEVEL) takes value between 0 and 1.

Example 14: (V-RELAX-LEVEL construct)

```
for $b in document(“bib.xml”)//book
where $b/year > 2000 V-RELAX-LEVEL 0.5
return $b
```

In the query above, V-RELAX-LEVEL specifies the maximum value relaxation level for book’s year to 50%.

Example 15: (S-RELAX-LEVEL construct)

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] S-COND-ALIAS c1 S-RELAX-LEVEL 0.5 and
($p/year > 2000) VC-ALIAS c2 V-RELAX-LEVEL 0.7
return <answer> {$p/title, $p/year}</answer>
```

12.2.2 Global Relaxation Control Operator

Relax-Order

The RELAX-ORDER construct can be used to control the order of relaxation explicitly if there are multiple relaxable conditions. The syntax for the RELAX-ORDER is as follows:

RELAX-ORDER '[' A {B (C D)} E ..']'

where A (B, C, D or E) can be a condition alias, an xtah name, a tah name, a variable name as defined as the FOR/LET clause, or a variable name with a list of relaxed binding expressions. We will explain the last case later. A condition must be tagged with an alias first before a relaxation order can be specified on it. The VCOND-ALIAS (S-COND-ALIAS) operation can be used to tag the value (or the structure) part of the conditions. There are three types of ordering:

- Conditions specified within the squarely brackets means absolute order. For example, [c1, c2] means c1 is relaxed before c2 is relaxed.

- Conditions specified within the regular parenthesis means same order. For example, (c1, c2) means both c1 and c2 have to be relaxed simultaneously.

- Conditions specified within the curly brackets means no order. For example, {c1, c2} means it does not matter whether c1 or c2 gets relaxed first. The system will use its default heuristics to pick which one to relax first.

  Examples 16 & 17:

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] USE-XTAH t1 and ($p/year > 2000) USE-TAH t2
return <answer> {$p/title, $p/year} </answer>
RELAX-ORDER [t1, t2]

for $b in document (“dblp.xml”)//book
where $b/year > 2000 S-COND-ALIAS c1 and $b/price <= $50 V-COND-ALIAS c2
return $b
RELAX-ORDER [c1, c2]
```

In the RELAX-ORDER clause, we can also specify a list of relaxed binding expressions for a variable, with the syntax as such:

$\$\text{VarName} <\text{RBE}_1, \text{RBE}_2, \ldots, \text{RBE}_n>$

where RBE_i stands for the i^{th} relaxed binding expression. For example, the query bellow finds addresses in USA with zip code approximately equal to 90095. If there are insufficient answers available, we can relax the value for the address’s zip code first, then relax the binding expression for the variable $a to “/order/*/address” and “//address” in order.

Example 18: (RELAX-ORDER construct with list of relaxed binding expressions)

```
for $a in document(“order.xml”)/order/customer/address
where $a/country = “USA” and $a/zipcode = ~ 90095 V-COND-ALIAS t1
return $a
RELAX-ORDER [t1, $a</order/*/address, //address>]
```

At-Least

The AT-LEAST construct specifies the minimum number of XML answers returned from the query that a user wants. If sufficient number of exact answers exists, no relaxation needed. Otherwise, the relaxation process begins.

Example 19:

```
for $p in document (“dblp.xml”)//paper
where $p/title [contains (., “Hidden Web”)] USE-XTAH t1 and ($p/year > 2000) USE-TAH t2
return <answer>{$p/title, $p/year}</answer>
RELAX-ORDER[t1, t2]
At-Least 5
```

Rank-By

The syntax is as such:

RANK-BY T1 [, weight$_1$], ..., T_N [, weight$_N$] [WITH method]

where T_i is either a condition alias or a XTAH/TAH name, and the weight is between 0 and 1. The rank-by operator is to specify that results are ranked by the specified closeness measure. The returned result should also include the score/ranking of the results.

Example 20:

The following query illustrates how the Rank-By clause is used. The query finds at least 5 DBLP papers published after year 2000 with title containing “Hidden Web”. If there are no exactly matched answers or not enough exactly matched answers, it performs structure relaxation on path expression paper/title first, and then value relaxation on $p/year. The returned results are ranked by the following measure:
Dist(Value($p/year), 2000)*0.7 + Dist(Structure($p/title), paper/title)*0.3

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] USE-XTAH t1 and ($p/year > 2000) USE-TAH t2
return <answer>{$p/title, $p/year}</answer>
RELAX-ORDER [t1, t2]
At-Least 5
RANK-BY [(t2, 0.7), (t1, 0.3)]
```

USE

The syntax is as such:

USE [st$_1$, … , st$_n$]

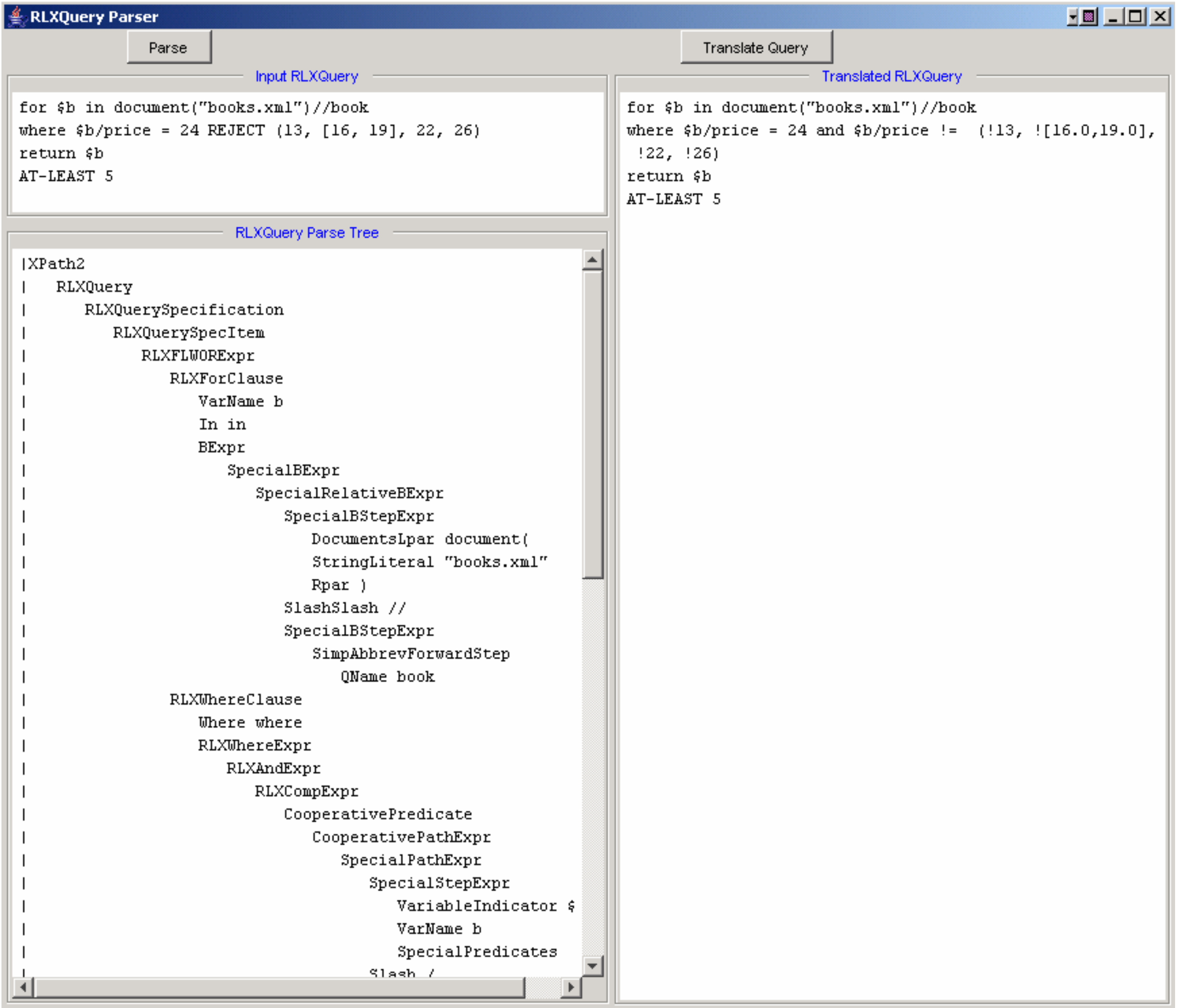
where st$_i$ stands for a structure relaxation type that can be used during the query relaxation process. We currently support four kinds of structure relaxation types: node re-label, node deletion, edge relaxation, and order relaxation. The USE clause allows users to specify structure relaxation types to be used during query relaxation process. For example, in the query below, relaxation types such as node re-label and edge relaxation are allowed, but not node deletion and order relaxations. When there is no USE clause, all four types of structure relaxation are assumed to be used.

Example 21:

```
for $p in document (“dblp.xml”)//paper
where $p/title[contains(., “Hidden Web”)] USE-XTAH t1 and ($p/year > 2000) USE-TAH t2
return <answer>{$p/title, $p/year}</answer>
RELAX-ORDER [t1, t2]
At-Least 5
RANK-BY [(t2, 0.6), (t1, 0.3)]
USE [RELABEL, EDGE]
```

13 Current Implementation

Currently the parser and the preprocessor have been successfully merged, and a graphical user interface (GUI) has been integrated on top of these two components. The functionality of the current project allows a user’s query in RLXQuery form to be fully translated to standard XQuery form[5]. Below is a screen shot of the GUI with a user’s input query.



```
for $b in document("books.xml")//book
where $b/price = 24 REJECT (13, [16, 19], 22, 26)
return $b
AT-LEAST 5
```

```
for $b in document("books.xml")//book
where $b/price = 24 and $b/price !=  (!13, ![16.0,19.0],
 !22, !26)
return $b
AT-LEAST 5
```

```
|XPath2
|   RLXQuery
|      RLXQuerySpecification
|         RLXQuerySpecItem
|            RLXFLWORExpr
|               RLXForClause
|                  VarName b
|                  In in
|                  BExpr
|                     SpecialBExpr
|                        SpecialRelativeBExpr
|                           SpecialBStepExpr
|                              DocumentsLpar document(
|                              StringLiteral "books.xml"
|                              Rpar )
|                           SlashSlash //
|                           SpecialBStepExpr
|                              SimpAbbrevForwardStep
|                                 QName book
|               RLXWhereClause
|                  Where where
|                  RLXWhereExpr
|                     RLXAndExpr
|                        RLXCompExpr
|                           CooperativePredicate
|                              CooperativePathExpr
|                                 SpecialPathExpr
|                                    SpecialStepExpr
|                                       VariableIndicator $
|                                       VarName b
|                                       SpecialPredicates
|                                    Slash /
```

The results of a valid RLXQuery query after preprocessing. Here the REJECT clause is translated.

[5] The one exception is a PREFER clause, which is handled by the relaxation controller.

The GUI allows a user to easily enter his query and see it get processed in each step of the project. First a user can type in his query in the "Input RLXQuery" text box. The "Parse" button will invoke the parser and send the input text query to it. The parser will return the results of the parsed query, either as an error message upon an invalid query or as a parse tree upon a valid query. The results from the parser will be displayed in the "RLXQuery Parse Tree" text box. If the query was valid, the XQueryRepConvertor automatically creates the XQueryRep object from the parse tree. This underlying implementation, however, is abstracted from the user. After receiving a valid query, the "Translate Query" button becomes enabled. This button invokes the preprocessor by sending it the underlying XQueryRep object for preprocessing. After the preprocessor does its work, the translated query will be displayed in the "Translated RLXQuery" text box.

It is also worthy to note that for the purposes of testing the project, a simple TAH Manager and XTAH Manager were created. Only the TAH Manager was needed by the preprocessor. The TAH Manager is basically a dummy implementation that returns a few hard-coded approximate values given a particular approximate term.

14 Future Work

A significant part of the RLXQuery engine has already been completed. Although the relaxation itself has not been implemented, much of the groundwork has been set. One important thing to remember is that the data structures for the RLXQuery language have been continually modified and have now reached a fairly robust state. The XQueryRep object can represent every query that we have thought of. Not only are data structures established, but we know how to use them. The work in the XQueryRepConvertor and in the preprocessor has given us complete knowledge on how to use the XQuery Rep object. Another significant aspect of the current project is how everything is modularized based on functionality and then linked up with the GUI. The GUI is controlling everything and making the appropriate calls to the different modules. Lastly, the XQueryRepConvertor is a significant player in the project so far. This class does all the dirty work in creating the XQueryRep object automatically from a parse tree. Creating an XQueryRep object manually is a very long process. The following is list of future work.

14.1 Relaxation Controller

The relaxation controller is the next module to implement. This module will likely be the most difficult to implement. Extensive use of the XTAH and TAH Managers is required, in conjunction with understanding the many different types of relaxation constructs and their uses.

Basically, the relaxation controller will execute the query the instant it gets the translated query from the preprocessor. (The one exception is when there is a PREFER construct, which the relaxation controller will have to translate to the first value in the value list.) Based on the number of answers returned from the database, the query might need to be relaxed. If so, at this point, the relaxation controller needs to determine which conditions it needs to relax first and in what order. This involves determining what structures (and/or values) are *not* declared non-relaxable and then looking for specifically tagged structures (and/or values). Then any global relaxation operators will need to be taken into account, which can determine the relaxation threshold, the order in which to relax, and the types of allowable structural relaxation. All of these factors together will make the relaxation controller a challenging component.

14.2 TAH / XTAH Manager

A TAH Manager has already been implemented in the CoBase project. We are looking to adapt this TAH Manager to an XML database implementation. An XTAH manager has not yet been created anywhere. Much work still needs to be done in this area.

14.3 Execution Trace Manager

The Execution Trace Manager will be a nice additional component to have in the RLXQuery engine. It will keep track of all the changes performed on the XQueryRep object by the relaxation controller. A user will be able to query the Execution Trace Manager about specific changes made to the XQueryRep object. All modules need to communicate with the Execution Trace to record their changes to the XQueryRep. When the relaxation controller receives the XQueryRep object from the parser, it will initialize an Execution Trace Object.

15 Conclusion

Query relaxation can be used in a number of different applications. The development of the RLXQuery language is significant because it makes use of the growing XML format. RLXQuery not only relaxes a query's constraint values, but it also extensively manipulates the path expressions used to formulate an XML query, which is important as the schema of XML databases have become increasingly more complex. The addition of an XTAH Manager allows for the relaxation of path expressions.

The parser and the preprocessor have been completed with the help of a set of RLXQuery object classes that allow an RLXQuery query to be stored as an object. The parser translates the user's query string into an XQR object, and the preprocessor takes the XQR object and translates any RLXQuery syntax to standard XQuery syntax. The next step in building the system is the relaxation controller, which will serve as the central component of the RLXQuery engine. The TAH and XTAH managers will assist the relaxation controller as the "knowledge sources."

There is still quite a bit of work to be done before the RLXQuery language can be tested in any application. The relaxation controller will require a significant amount of time in development. The XTAH Manager will also require some work, and the TAH Manager will be to be adapted from the CoBase implementation. Once complete, the RLXQuery system can be substituted in for any XML database system.

16 References

1. Shaorong Liu and Wesley W. Chu. Cooperative XML (CoXML) Query Answering at INEX 2003. In *Proceedings of the 2nd INitiative of the Evaluation of XML retrieval (INEX) Workshop*, Schloss Dagstuhl, Germany,2003.
2. Eric Sung. Parser for Relaxation-Enabled XML Query Language. *Masters Thesis*, Computer Science Department, UCLA, Fall 2004.
3. Anna Putnam. The Relaxation Manager for XML Query Relaxation. *Masters Thesis*, Computer Science Department, UCLA, Spring 2004.
4. Shaorong Liu, "XML Query Relaxation Constructs"
5. CoBase Project, http://www.cobase.cs.ucla.edu/
6. Galax Website, http://db.bell-labs.com/galax/
7. XQuery, http://www.w3.org/TR/xquery/

17 Appendix

Please refer to the following web site for the source code:

www.cobase.cs.ucla.edu/projects/CoXML/doc/pre-relaxation/