CXLENGINE - A COMPREHENSIVE XML LOOSELY STRUCTURED SEARCH ENGINE

by

INDHU KRISHNA SIVARAMAKRISHNAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2007

ACKNOWLEDGEMENTS

At the outset, I would like to thank Dr.Ramez Elmasri for giving me an opportunity to work with him. I would also like to thank Dr. Gautam Das and Dr. Leonidas Fegaras for agreeing to be on my master's thesis committee and helping me complete this work.

Next, I would like to thank my parents and sister for their continuous support and words of encouragement.

My heartfelt gratitude to Mr. Kamal Taha for his invaluable suggestions and constant guidance throughout this work.

Finally, I would like to thank my friends for always being there for me.

November 19, 2007

ABSTRACT

CXLENGINE - A COMPREHENSIVE XML LOOSELY STRUCTURED SEARCH ENGINE

Publication No. _____

Indhu Krishna Sivarmakrishnan, M.S

The University of Texas at Arlington, 2007

Supervising Professor: Dr. Ramez Elmasri

XML keyword search has been a widely researched area. [15] has proposed an XML semantic search engine called OOXSearch, which answers loosely structured queries. The framework of OOXSearch takes into account the semantic relationship between nodes based on their contexts. The context of a node is determined by its parent node. The label "name", for example, could mean the name of a book and could also mean the name of the book's author. If we try to consider the relationship between these two nodes without considering their parent nodes, we would be drawing the incorrect conclusion that these two nodes represent the characteristics of two entities belonging to the same type. Thus, the OOXSearch framework treats a parent and its

leaf nodes as a single unified entity. The OOXSearch Engine works well for all types of XML trees, except for one scenario where a parent and its child node belong to the same type and are both having leaf children nodes. In this thesis, we propose an extension to the OOXSearch Engine that handles the above mentioned case. We experimentally evaluated the extended search engine and compared the results with other systems.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Keyword Search is a widely used search technique since it has the advantage that the user need not be aware of the structure of the data underneath. One more reason for using keyword search is that the user need not know any query language in order to be able to get the required data.

There has been lots of research related to keyword search in databases. Research in the area of keyword search in relational databases was studied extensively in [3], [4] and [11]. These papers considered the relational database as a graph with the edges representing the relationships between the nodes where the nodes represent the tuples in the database. The result is returned as a sub-graph which contains the keyword search terms.

There has been extensive research on Keyword Search in XML databases. Out of these, the papers most related to our thesis are [9], [12] and [18] which employ search techniques based on semantic relationships between nodes. The basic notion shared by these papers is the concept of the Least Common Ancestor (LCA) of the search term nodes.

A closely related work is [15]. In this paper, an XML search engine is proposed that answers loosely structured queries. The search queries consist of search terms of the form “label=keyword” and return elements have the form “label”. For example, a

user wanting the year a book named “JAVA” was written could submit a query of the form “name=JAVA” and the return element will be of the type “year”. The motivation behind [15] is that the context of a node should be taken into consideration while considering the semantic relationship between nodes. For example, two nodes might be having the same label “name”. But one label could be referring to the name of a book while the other one could be referring to the name of the author who wrote that book. The authors of [15] reiterate that the parent node’s characteristics will also need to be taken into consideration. Thus, the context of a node is determined by its parent node. [15] considers the parent and its leaf children nodes as a single unified entity. This entity is termed a Canonical Tree. For example, consider the nodes title and year in [Figure 1.1]. They represent some of the characteristics of their parent node article. The nodes article, title and year together represent a meaningful entity. Each such entity can be modeled as a Canonical Tree. There could be some confusion due to the labeling. For example, there could be two different entities with the same label – name – where the name could refer to either the name of the book or the name of the author who wrote that book. One more case where there could be some ambiguity is when there are two entities which have different names but belong to the same type. For example, there could be 2 entities with the names “school” and “university”. Both belong to the type “institution”. To avoid these labeling ambiguities, the authors of [15] introduced the term clusters which are nothing but groups of canonical trees where the parent nodes belong to the same ontological concept.

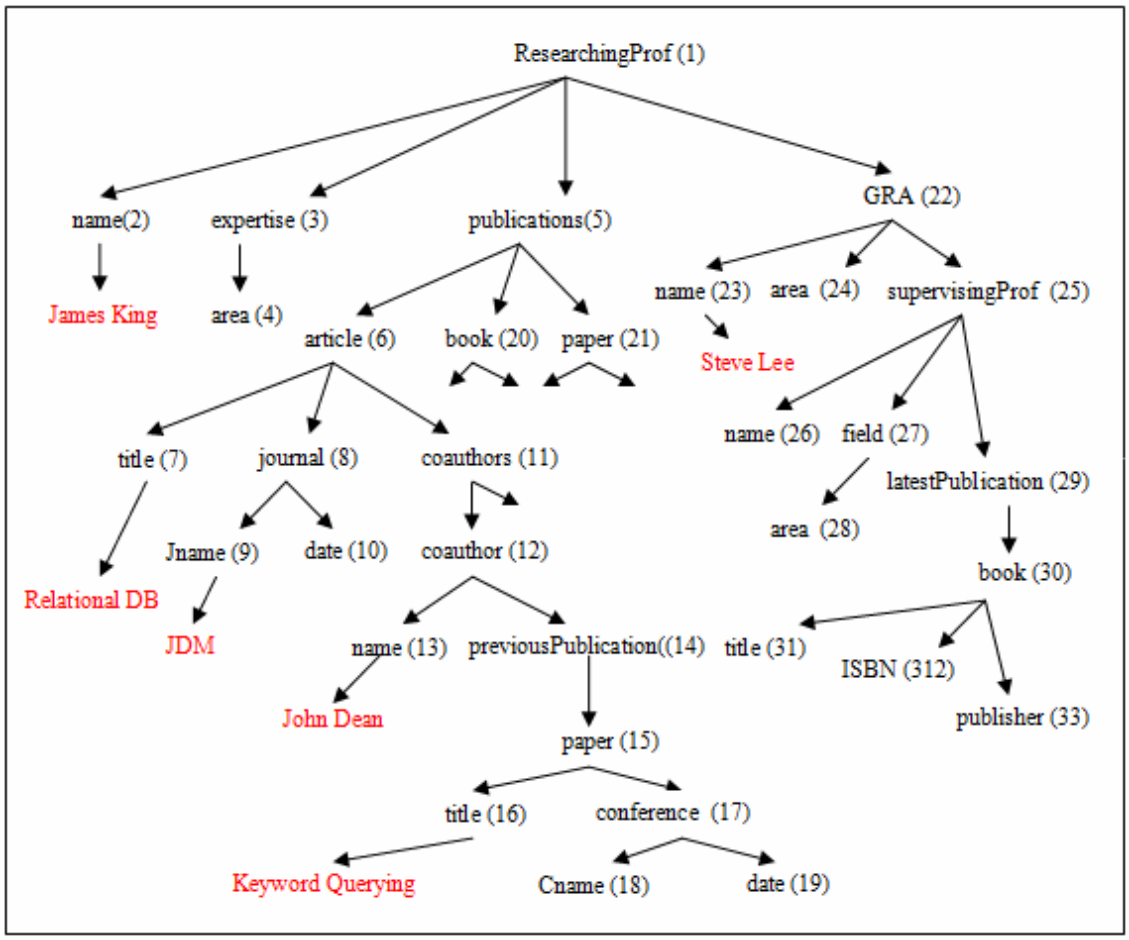

Figure 1.1 : Graduate School Publications XML tree (publication.xml)

For example, if we take the canonical tree rooted at “Student” and the canonical tree rooted at “Professor”, both of them belong to the “person” cluster. If we take the canonical tree rooted at “School”, it will belong to the “Institution” cluster. Thus, even though both the trees rooted at “Student” and “School” have a node with the same label – “name” – we can see that these two name nodes are semantically different as they

represent a characteristic of two entities of different types. The main conclusions of the study in [15] are as follows:

- Modeling an XML scenario in terms of canonical trees helps us in finding for each canonical tree T_i the canonical trees that are related and relevant to T_i in a most efficient manner.
- If canonical tree T_j is related and relevant to canonical tree T_i, it follows that all the leaf children nodes of T_j are also related and relevant to the leaf children nodes of T_i. Thus computational overhead is reduced.
- The structure of canonical trees can be considered analogous to the Object Oriented Model. Each canonical tree can be modeled as an object and its nodes can be the attributes of the object. Hence, we can construct a class for each canonical tree. We can also model the concept of inheritance in Object Oriented concept by designing the Ontological label of the cluster to be the superclass. Thus, each canonical tree under this cluster can be a subclass. For example, both the "School" and "University" classes can be subclasses of the superclass "Institution" and can inherit the "name" attribute from the superclass.

The OOXSearch model works well for all types of XML trees except for one case where a parent and its child node belong to the same type and are both having leaf children nodes. In this thesis, we propose to extend the OOXSearch framework so that it works for the above mentioned case. The motivation behind this work is to make the OOXSearch framework a fully fault-free one.

CHAPTER 2

XML

2.1 Origin of XML

XML stands for "Extensible Markup Language". It is basically a markup language. As users can specify their own tags, it is also termed an extensible language. It started as a subset of Standard Generalized Markup Language (SGML). The work began in 1996 and Version 1.0 Recommendation was released in 1998. XML is recommended by World Wide Web Consortium (W3C). The formal definition from W3C glossary at http://www.w3.org/TR/DOM-Level-2-Core/glossary.html is as follows:

"Extensible Markup Language (XML) is an extremely simple dialect of SGML. The goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML. "

XML is a way to store structure and metadata onto a web page. It allows users to specify tags which make the information stored more powerful and flexible.

2.2 Why XML

This section outlines with the reasons for XML's popularity. XML is widely used for Web applications. There are various benefits to working with XML. Some of them are as follows:

- *Simplicity*

  Information specified in XML is easy for human and computer understanding and processing.

- *Extensibility*

  The tag set is not fixed in XML. New tags can be created as and when needed.

- *Self-description*

  XML documents do not need any schema information to be stored as the XML documents themselves contain metadata. Metadata is stored in XML via tags and attributes.

- *Openness*

  XML is recommended by the World Wide Web Consortium (W3C). It is a fee-free open standard and is widely endorsed by the industry.

- *Machine Readable Context Information*

  The context information in XML is provided by means of the tags and attributes and these can be used to arrive at the content information of

the data. This feature enables for the development of intelligent data mining techniques.

- *Support multiple data types*

  Any data type like sound, image, video, JAVA applets, etc can be stored in XML documents.

- *Portability*

  XML documents can be shared between different systems like PC, PDA, etc without loss of information.

- *Interoperability*

  One of the main benefits of XML is its interoperability which makes it possible for disparate systems to share information easily.

2.3 XML Syntax

This section deals with the syntax of XML documents. Before looking at the syntax, there are two important definitions to be covered. XML documents can have the following 2 levels of correctness:

- *Well-formed documents*

  Well-formed XML documents are those that conform to the syntax rules. For example, in a well-formed document, all the opening tags are matched with an equal number of closing tags and there is no case such that an outer element has its closing tag appearing before the closing tag of the inner nested element.

- *Valid documents*

  A valid XML document conforms to a set of semantic rules. These rules could be specified by means of a Document Type Definition (DTD) or could be specified by the user. Valid documents will contain no undefined tags.

Most XML documents an XML declaration. This specifies the version of XML and sometimes, also specifies the character encoding used.

<?xml version="1.0" encoding="UTF-8"?>

As XML documents are hierarchical in nature, the main requirement is that there is exactly one *root element*. The child elements are nested inside the root element. The elements could also have *attributes*. A typical element with attributes would look like this:

<element name attribute1 = "attribute value" attribute2 = "attribute value" …..…

attribute n = "attribute value" > element text </element name>

Attribute values must be given in double quotes and an attribute name can appear in an element only once.

A sample XML document is given below:

```
<?xml version="1.0"?>
<!DOCTYPE department SYSTEM "bib.dtd">
<department>
      <author >
            <name>Mary Jones</ name>
            <URL>www.mjones.org</URL>
            <address>SanFrancisco,
                        CA</address>
       </author>
</department>
```

Figure 2.1 Sample XML Document

2.4 DTD

DTD stands for *Document Type Definition*. DTD represents the schema of an XML document. It defines the structure of the XML document with a set of allowed attributes and elements. A DTD is associated with an XML document by using a Document Type Declaration. This usually comes right after the XML declaration in the XML document. An example declaration is follows:

<!DOCTYPE department SYSTEM "filename.dtd">

An example DTD is as follows:

```
<!ELEMENT department (author+)>
<!ELEMENT author (name,URL,address)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT URL (#PCDATA)>
<!ELEMENT address (#PCDATA)>
```

Figure 2.2 Sample DTD

The explanation for the DTD is as follows:

- The first line specifies the root element – department and the ELEMENT tag specifies that this is an element node.
- The data following the root element is the name of the element nested inside the root element – author.
- The (+) means that it is valid to have 1 or more “author” elements inside the “department” element.
- The second line gives the schema of the “author” element. It specifies that it has 3 elements nested inside – “name”, “URL” and “address”.
- The next 3 lines give the data type of the name, URL and address elements. PCDATA specifies a string data type.

2.5 Querying in XML

There are a wide variety of query languages used to query XML documents. Some of these are XPath, XQuery, XQL, YaTL, etc. As XPath and XQuery have already been recommended by W3C, those two will be the focus in this thesis.

2.5.1. XPath

XPath is a query language used to extract parts of the XML document according to user specifications. XPath models the XML document as a set of nodes – element nodes, text nodes, etc. The basic premise used in an XPath query is an *expression*. Every expression evaluates to one of the following types:

- *NodeSet* – an unordered collection of nodes with no duplicates
- *Number* – a floating point number
- *String* – a sequence of characters
- *Boolean* – TRUE or FALSE

XPath uses path expressions to navigate in XML documents. A path expression is represented as a sequence of steps to arrive at a node or a set of nodes from the current node. The series of steps are delimited by '/'. A typical step has the following three components:

- *Axis Specifier*

  The axis specifier gives the direction in which the navigation has to proceed. Some example axis specifiers are *child, descendant (//), parent (..), ancestor, following, preceding, self*, etc.

- *Node Tests*

  Node tests contain node names or general expressions.

- *Predicates*

  Expressions are specified in square brackets and these have to evaluate to TRUE before the preceding node can be matched. There can be an unlimited number of predicates in a step.

An example XPath expression is as follows:

//department[@name=”CSE”]/student[@gpa=”4.0”]/lastname

The above XPath Query gives the last name of the students in the CSE department with a GPA of 4.0.

2.5.2. XQuery

XQuery is an extension of XPath. It adds to XPath with a construct similar to SQL. This is the FLWOR expression and is named for its clauses – FOR, LET, WHERE, ORDER BY and RETURN. XQuery also allows XML syntax which helps in the creation of new XML documents. If the element and attribute names are known in advance, tags can be added to the result. As XQuery is similar to XPath in a lot of aspects, it too models the XML document as a collection of nodes. The nodes could be element nodes, attribute nodes, text nodes, etc. An example XQuery query is as follows:

for $b in doc("bib.xml")//book

where $b/b_ISBN = "0-9752298-0-X"

return $b/b_price

The above query returns the price of the book whose ISBN number is “0-9752298-0-X”.

CHAPTER 3

KEYWORD BASED SEARCH

Keyword search is a widely researched area. The main advantage of keyword search is that the user is not required to know the schema of the underlying data. Also, it is not necessary to have knowledge of any query language. Both the above points make keyword search a major success with end users. Keyword search has been studied for both relational databases and for XML databases.

3.1 Keyword Search for Relational Databases

Research in the area of keyword search in relational databases was studied extensively in [3], [4], [11] and [23]. These papers considered the relational database as a graph with the edges representing the relationships between the nodes where the nodes represent the tuples in the database. The result is returned as a sub-graph which contains the keyword search terms. While [3], [11] and [23] construct a set of join expressions from the database and then form the tuple trees from these join expressions, [4] creates the tuple trees directly from the database. [3], [4] and [11] use AND semantics for the result whereas [23] uses OR semantics.

3.2 Keyword Search for XML Databases

There has been extensive research on Keyword Search in XML databases. [9], [12] and [18] employ search techniques based on semantic relationships between nodes. The above mentioned studies concentrate on the computation of the Least Common

Ancestor (LCA) of the search term nodes. [9] proposed the interconnection relationship between a set of nodes n1,n2,…..,nk which specifies that the relationship tree of n1, n2, ….., nk does not contain two nodes with the same label or that the only nodes with the same label in the relationship tree of n1, n2, ….., nk are among n1, n2, ….., nk. [12] introduced the concept of Meaningful Lowest Common Ancestor (MLCA). The MLCA of a set of nodes it the most specific XML structure in which the nodes are related. Two nodes "a" and "b" are meaningfully related if their LCA (Lowest Common Ancestor) node 'c' is not an ancestor of some other node 'd' which is an LCA of node "b" and some other node that has the same label as node "a". [18] came up with the notion of Smallest Lowest Common Ancestor (SLCA). The SLCA of keywords is a set of nodes that contain the keywords either in their labels or in the labels of their descendant nodes and there are no descendant nodes that also contain all the keywords.

[6] and [10] concentrate on modeling the XML document as a graph. [6] considers the XML tree as a labeled graph and the edges correspond to the element-sub element relationships and the IDREFs. The XML data is stored in a relational database for fast and efficient access which leads to smaller response times. The key contribution of this paper is in the prevention of information flow and the user-interface feature where users are allowed to navigate in the result. The user can expand the available result so as to see more relevant results. [10] models the XML scenario as a rooted directed graph. Nodes are represented as objects that have a label (also a value, sometimes). Edges represent element nesting and IDREFs. The authors of [10] propose 2 types of interconnection semantics – implicit (derived) and explicit. Users can specify

the interconnection semantics explicitly by manually specifying the patterns. In the implicit way of defining the interconnection semantics, conditions that characterize the pattern can be specified.

A number of recent studies [9, 12, 18] employ semantic search over XML docs modeled as trees, which makes them the closest to our work. Despite their success they suffer recall and precision limitations as a result of basing their techniques on building relationships between data nodes based solely on their labels and proximity to one another while overlooking their contexts. As a result, the proposed search engines may return faulty answers especially if the XML document contains more than one node having the same label but representing different types, or having different labels but belonging to the same type. We are going to demonstrate below the recall and precision limitations of each of the three proposed search engines by using samples of queries selected from the ones used in the experiments. Recall is the ratio of the number of relevant records retrieved to the total number of relevant records in the database, and precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. We will denote below each faulty recall by RF (which stands for Recall Fault) and each faulty precision by PF (Precision Fault).

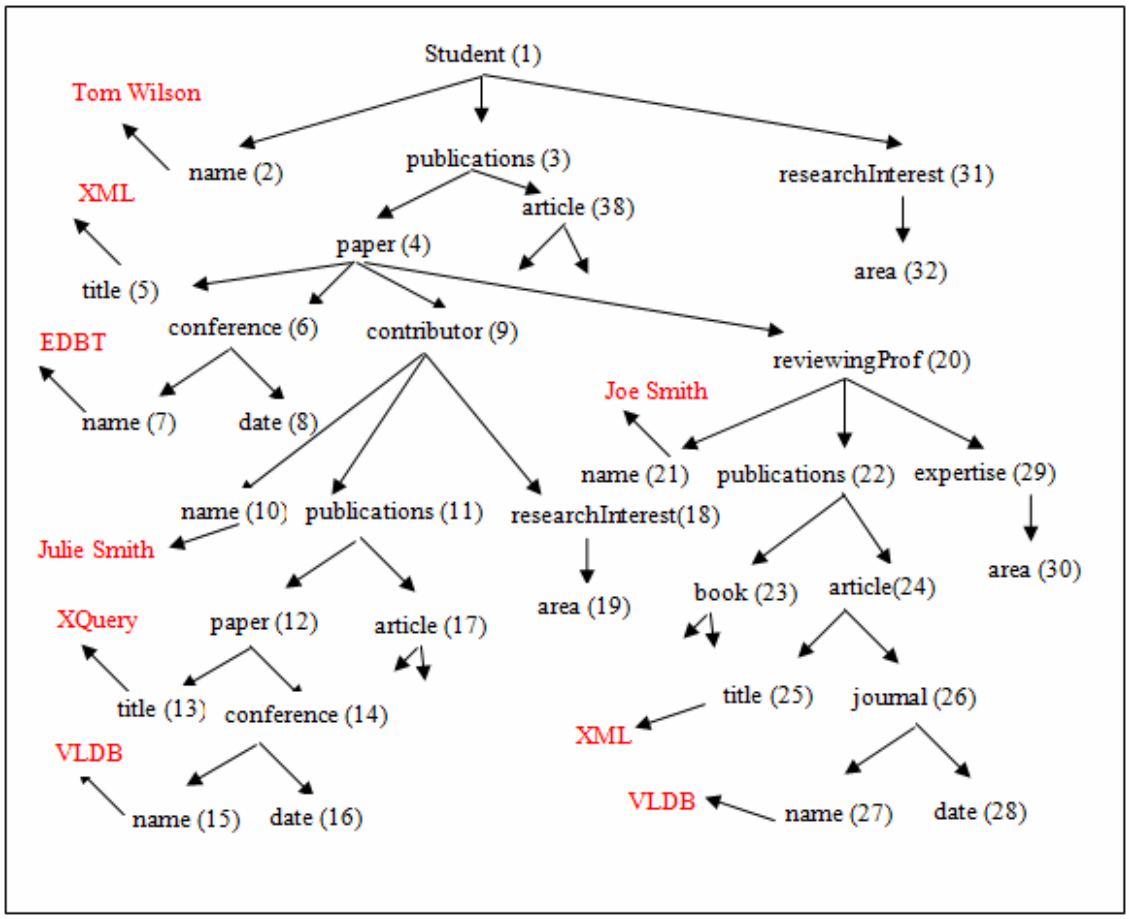

Figure 3.1: A graduate school's authors and coauthors bibliography XML tree

The paper (node 4) was authored by a student (node 1) and coauthored by a contributing student (node 9) and a reviewing professor (node 20). The publications (node 11) were authored by the contributing student (node 9 only). The publications (node 22) were authored by the reviewing professor (node 20) only. Note that the XML tree is exaggerated for the purpose of demonstrating all the concepts proposed in this thesis.

Recall and precision faults of XSEarch [9]:

In XSEarch if the relationship tree of nodes a and b (the path connecting the two nodes) contains two or more nodes with the same label, then the two nodes are unrelated; otherwise, they are related.

PF: Consider Figure 3.1 and the query "what is the research interest area of Tom Wilson". Instead of returning node 32 only, [9] will return also nodes 19 and 30, because the relationship trees of nodes 2 and 19 and of nodes 2 and 30 do not contain two or more nodes having the same labels. If [9] employs the ontological concepts we are proposing, it would have discovered that the first relationship tree contains nodes contributor and student which belong to the same type and the second relationship tree contains nodes reviewingProf and student which also belong to the same type.

RF: On the flip side, consider Figure 3.2 and the query "who is the coauthor of the publication authored by Julie". Instead of returning node 2 "Tom", [9] will return null, since the relationship tree of nodes 4 and 2 contains two nodes having the same label (nodes 1 and 3, which have the same label "author").

PF: Consider the XML doc in Figure 3.3 and the query "what is the customer's name, who placed an order on 5/20/06". Instead of returning "David" only, [9] will return also "UPS", because [9] can't determine that the two nodes labeled "name" refer to two entities having different types.

Recall and precision faults of Schema- Free XQuery [12]:

In [12], nodes "a" and "b" are NOT meaningfully related if their LCA, node 'c' is an ancestor of some node "d", which is a LCA of node "b" and another node that has the same label as "a". Consider for example nodes 2, 10, and 19 in Figure 3.1. Node 19 (area) and node 2 (name) are not related, because their LCA (node 1) is an ancestor of node 9, which is the LCA of nodes 19 and 10, and node 10 has the same label as node 2.

Therefore, node 19 is related to node 10 and not to node 2. Node 9 is considered the Meaningful Lowest Common Ancestor (MLCA) of nodes 19 and 10.

RF: Consider Figure 3.4 and the query "get the image presented in the section titled "Introduction". The correct answer is node 6, but [12] will return null. The reason is that the LCA of nodes 3 (which contains the keyword Introduction) and 6 is node 2, and node 2 is an ancestor of node 4, which is the LCA of nodes 6 and 5 (which has the same label as node 3). Therefore, [12] considers node 6 is related to node 5 and not to node 3.

RF: Consider if we prune node 30 (area) from Figure 3.1 and we have the query (what is the area of expertise of "Joe Smith"). Instead of returning null, [12] will return node 19 (area).

PF: Consider Figure 3.5 and the query "what is the work-shop's subject title on July 28". Instead of returning node 3 only, [12] will return also nodes 4 and 5, which are irrelevant (titles of papers). As another example, consider Figure 3.1 and the query (what are the publications' titles authored by "Tom Wilson"). [12] will return nodes 5, 13, and 25. But, nodes 13 and 25 are irrelevant.

RF: Consider Figure 3.1 and the query "who are the coauthors of the publication, where one of its authors is "Tom Wilson". The correct answer is nodes 10 and 21, but [12] will return null. The reason it does not return node 10 is because node 1 is the LCA of nodes 2 (which contains the keyword "Tom Wilson") and node 10 and it is also an ancestor of node 4, which is the LCA of nodes 10 and 21 and the label of node 21 is the same as the label of node 2. Therefore, [12] considers node 10 is related to node 21 and not to node 2 and that node 4 is the MLCA of nodes 10 and 21. Similarly and in the same token,

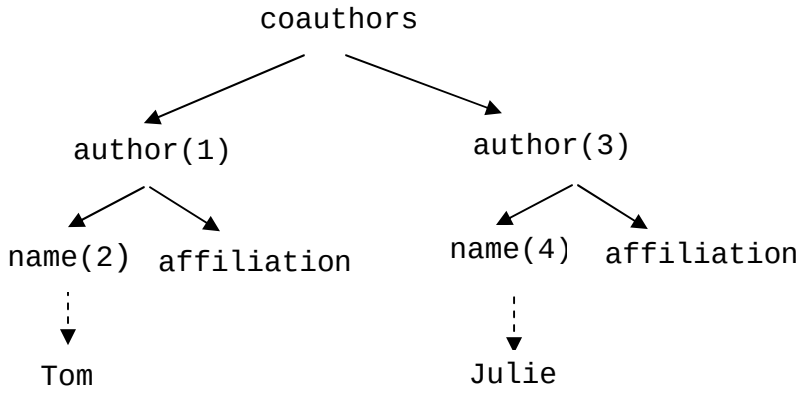

Figure 3.2: A fragment of XML doc taken from the web.

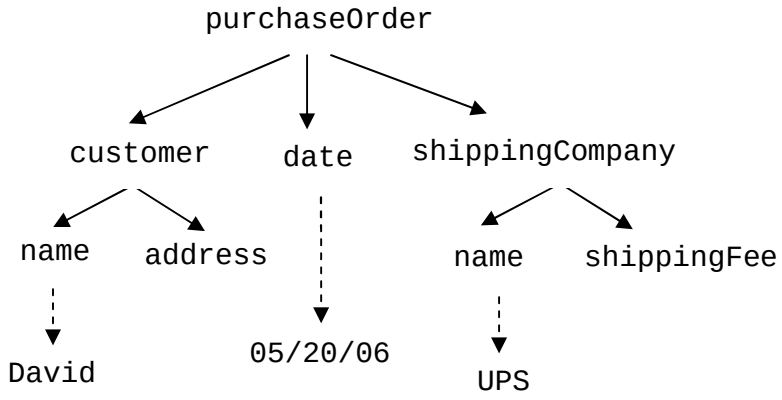

Figure 3.3: Taken from the Use Cases of W3C [19]

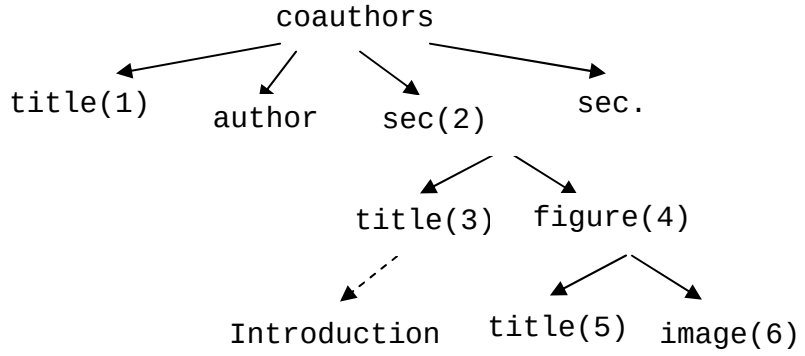

Figure 3.4: Taken from the Use Cases of W3C [19].

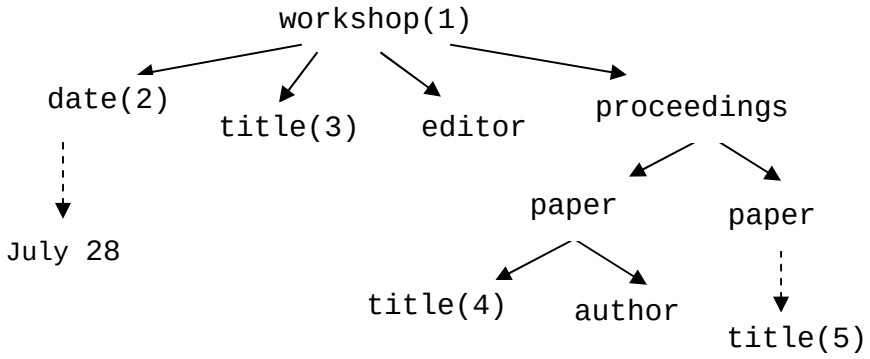

Figure 3.5: A fragment of XML doc taken from [8]

[12] does not return node 21, because it considers it related to node 10 and not to node 2.

Recall and precision faults of XKSearch [18]:

[18] returns a subtree rooted at a node called the Smallest Lowest Common Ancestor (SLCA), where the nodes of the subtree contain all query's keywords and they have no descendant node(s) that also contain all keywords. Consider for example Figure 3.1 and that node 13 contains "XML" instead of "XQuery". Now consider the query Q("XML", "Julie Smith"). Since the keyword "XML" is contained in both nodes 5 and 13, the answer subtree will be the one rooted at node 9 and contains nodes 10 and 13, and not the one rooted at node 4 and contains nodes 5 and 10.

RF: Consider Figure 3.1 and the query Q("Joe Smith", "XQuery"). The answer should be null, since Joe Smith didn't author nor coauthor the publication titled "XQuery". However, [18] will return the subtree rooted at node 4 and containing nodes 13 and 21.

PF: Consider Figure 3.1 and the query Q("Smith", "XML", "VLDB"). The query asks for information about an author, whose last name is "Smith" and who authored a publication titled "XML", which appeared in a "VLDB" conference proceedings or journal. [18] will return two answer subtrees, one is a correct answer and the other is incorrect. The first one is rooted at node 4 and contains the three keywords in nodes 5, 10, and 15. The second one is rooted at node 20 and contains the three keywords in nodes 21, 25, and 27. The first answer is incorrect, because the publication titled "XML" (node 5) and authored by "Julie Smith" was published in an "EDBT"

conference proceedings (node 7) and not in a “VLDB”, and her publication titled “XQuery” (node 13) is the one published in “VLDB” proceedings. The second answer is correct, because “Joe Smith” (node 21) authored a publication titled “XML” (node 25), which appeared in a “VLDB” journal (node 27).

PF: Consider Figure 3.1 and the query: Q(“Joe Smith”, “XML”, “EDBT”). [18] will return the subtree rooted at node 4. As can be seen the subtree contains too much irrelevant information.

CHAPTER 4

OVERVIEW OF OOXSEARCH

In this section, we present the key definitions and notations used in [15]. The XML documents are modeled as rooted and labeled trees. A node in the tree corresponds to an element in the XML document. Nodes are numbered for easy reference. This section also deals with the properties and lemmas used in [15].

The OOXSearch engine takes as input queries in the XQuery syntax. The basic prototype of a query is as follows:

for $d in doc("XML document name") where $d//node's label = "keyword"

return $d//node's label

Each label-keyword pair in the where clause is considered to be the search term and the label element in the return statement is the result element. The where clause may consist of more than one search term. Similarly, the return clause may also contain more than one result term. As mentioned already, the user need not be aware of the schema of the underlying XML document. He/she needs to know only the label and value of the search term and the label of the return element.

4.1 Definitions

In this section, we present the key definitions used in [15].

Definition 4.1 Ontology Label (OL):

If we cluster parent nodes (interior nodes) in an XML doc based on their reduced characteristics and cognitive qualities, the label of each of these clusters is an Ontology Label (OL). Table 4.1 shows the Ontology Labels and clusters of parent nodes in the XML tree in Figure 4.2. We abbreviate each OL to a letter called an Ontology Label Abbreviation (OLA). Table 4.1 shows also the OLA of each OL in the table.

Definition 4.2 Canonical Tree:

If we fragment an XML tree to the simplest semantically meaningful fragments, each fragment is called a Canonical Tree and it consists of a parent node and its leaf children data nodes. That is, if a parent node has a leaf child (or children) data node(s), the parent node along with its children data nodes constitute a Canonical Tree. In Figure 4.2 for example, the parent node student(1) and its leaf child data node name(2) constitute a Canonical Tree, whose ID is T_1 (see Figure 4.3). Similarly, the parent node paper(4) and its child data node title(5) constitute a Canonical Tree, whose ID is T_2. Since node paper(4) is a descendant of node student(1) and there is no interior node in the path between student(1) and paper(4) that has a child data node, therefore Canonical Tree T_2 is a child of Canonical Tree T_1 (see Figure 4.3). Leaf children data nodes represent the characteristics of their parents. The Ontology Label of a Canonical Tree is the Ontology Label of the parent node component in the Canonical Tree. For example, the Ontology Label of Canonical Tree T_1 in Figure 4.3 is the Ontology Label of the parent node student(1), which is “person”. Each Canonical Tree is labeled with two labels as the

Figure below shows. The label on top of the rectangle is the Ontology Label of the Canonical Tree. The second label is located inside the rectangle in the upper left side and has the form Ti, which represents the numeric ID of the Canonical Tree, where 1 i |T|. Figure 4.3 shows the Canonical Trees graph representing the XML tree in Figure 4.3. For example, the Ontology Label of the root Canonical Tree is "person" and its numeric ID is T_1. We sometimes use the abbreviation "C. Tree" to denote "Canonical Tree".

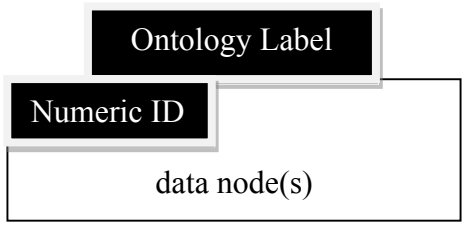

Figure 4.1: Canonical Tree Representation

Definition 4.3 Keyword Context (KC):

KC is a Canonical Tree containing one or more of a query's keywords.

Definition 4.4 Intended Answer Node (IAN):

When a user submits a keyword query, he is usually looking for data that is relevant to the query's keywords. Each one of the data nodes containing this data is called an Intended Answer Node (IAN). Consider for example Figure 4.2 and that a user submitted the query Q("Julie Smith", "VLDB"). As the semantics of the query implies,

the user wants to know information about the publication(s) authored by Julie Smith and appeared in a "VLDB" conference proceedings. This information is contained in nodes 13 and 16. Therefore, each of these two nodes is called an IAN.

Table 4.1: Ontology Labels and OLAs of parent nodes in Figure 4.2

Parent nodes (with their IDs)	**Ontology Label (OL)**	**OLA**
paper (4,12), article (17, 24), book (23)	publication	B
student (1), contributor (9), reviewingProf (20)	Person	P
researchInterest (18, 31), expertise (29)	Field	F
conference (6), journal (26)	proceedingsSponsor	S

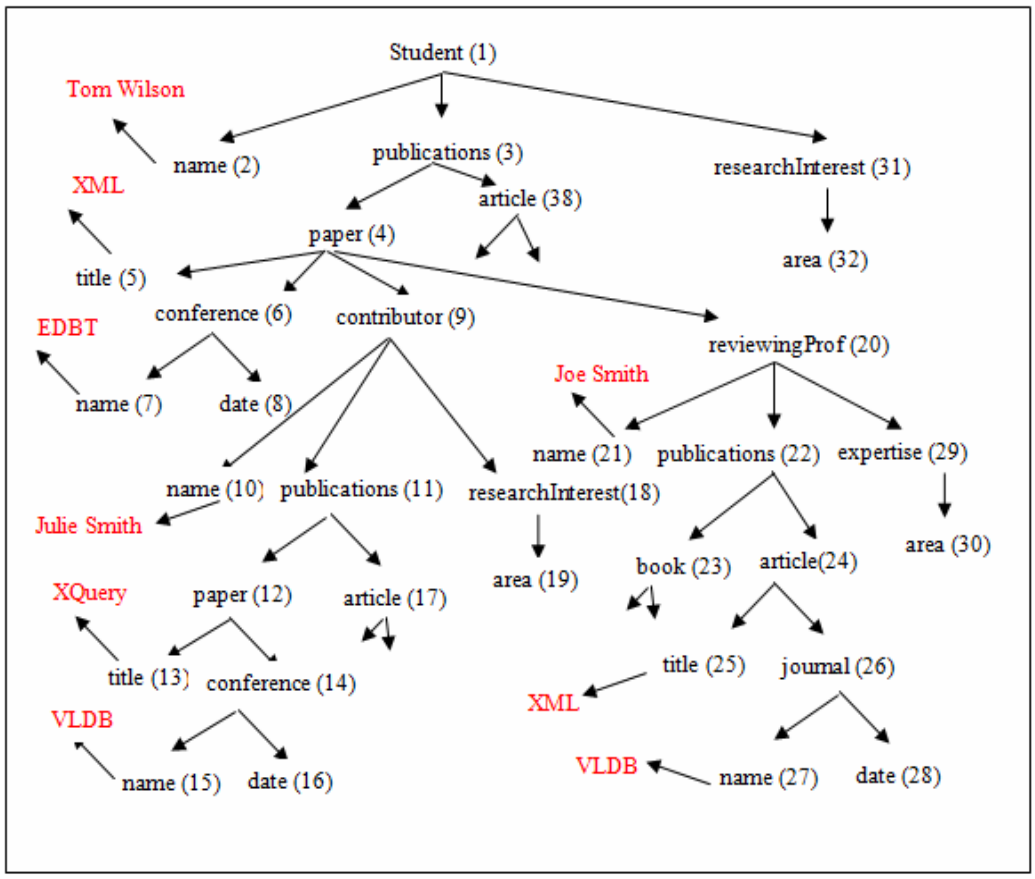

Figure 4.2: A graduate school's authors and coauthors bibliography XML trees

The paper (node 4) was authored by a student (node 1) and coauthored by a contributing student (node 9) and a reviewing professor (node 20). The publications (node 11) were authored by the contributing student (node 9 only). The publications (node 22) were authored by the reviewing professor (node 20) only. Note that the XML tree is exaggerated for the purpose of demonstrating all the concepts proposed in this thesis.

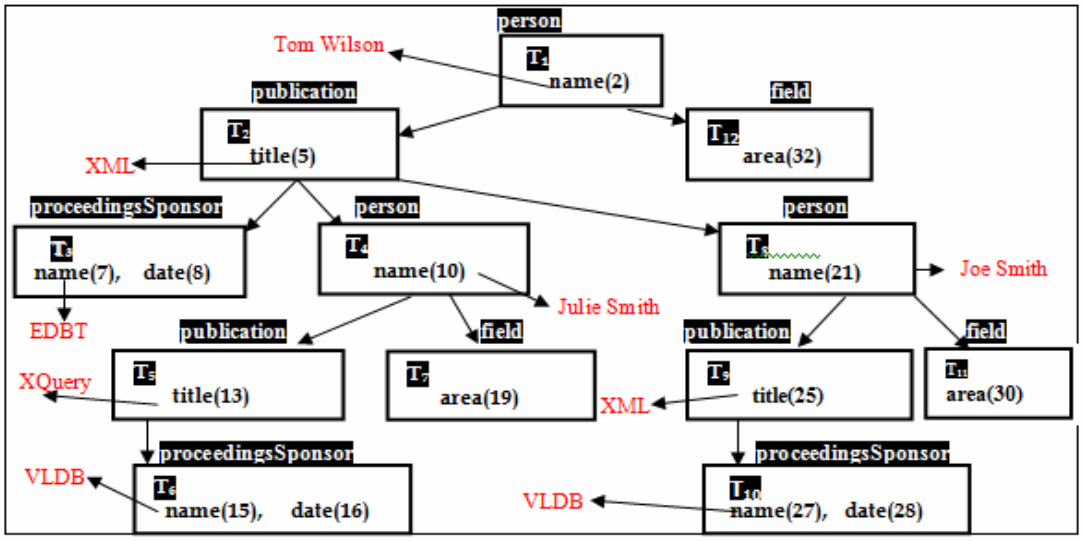

Figure 4.3: Canonical Trees graph of the XML tree presented in Figure 4.2

4.2 Computation of Immediate Relative of Canonical Trees

In this section, we present the key definitions used in [15].

Each Canonical Tree represents a real-world entity. A Canonical Trees graph (see Figure 4.3) depicts the hierarchical relationships between the entities represented by the Canonical Trees. A relationship between two Canonical Trees could be described as either immediate or extended. The Immediate Relatives of C.Tree T_i are Canonical Trees that have strong and close relationships with T_i, while its Extended Relatives have weak relationships with it.

We have to distinguish between two types of queries, query type A and query type B. In query type A, the search term element's label is different than the return element's label (e.g. the search term element is "name" and the return element is "title"). In query type B, both the search term element and the return element have the same label (e.g. the search term element is "name" and the return element is "name").

Notation 4.1 IR_T:

IR_T denotes the set of Canonical Trees that are Immediate Relatives of Canonical Tree T.

Notation 4.2 OL_T:

OL_T denotes the Ontology Label of Canonical Tree T. In Figure 4.3 for example OL_{T_1} is "person".

If a query's keyword k is contained in Canonical Tree T, the IAN should be contained in either T itself or in IR_T. If a query's keywords k_1 and k_2 are contained in Canonical Trees T_1 and T_2 respectively, the IAN should be contained in T_1, in T_2, and/or in the intersect $IR_{T1} \cap IR_{T2}$. Therefore, for each Canonical Tree T_i, we need to determine its Immediate Relatives (IR_{Ti}).

Determining IR_{Ti} could be done using the combination of intuition and logics that govern relationships between real-world entities. Consider for example Figures 4.2 and 4.3 and the query Q(XQuery). It is intuitive that the IAN to be data nodes 10, 15, and/or 16 but it is not intuitive to be, for instance, node 2, since "Tom Wilson" (node 2) didn't author the publication "XQuery". Since "XQuery" is contained in Canonical Tree T_5, then we can determine that each of the Canonical Trees containing nodes 10, 15, and 16 $\in IR_{T5}$ while the Canonical Tree containing node 2 $\notin IR_{T5}$. We present below three lemmas that help in determining IR_{Ti} for query type A. Their proofs are based on intuition and logics that govern relationships between real-world entities. We are going to sketch the proofs. We can determine IR_{Ti} by pruning all the Extended Relatives of T_i from the Canonical Trees graph and the remaining ones would be IR_{Ti}. We present three properties inferred from the conclusions of the three lemmas, which regulate the pruning process. We also present a fourth property for the queries of type B.

Lemma 1:

In order for the answer of a keyword query to be intuitive and meaningful, the IAN should never be contained in a Canonical Tree whose Ontology Label is the same as the

Ontology Label of the KC (the Canonical Tree containing the keyword). It could be contained in either the KC itself or in a Canonical Tree whose Ontology Label is different than the Ontology Label of the KC. Therefore, the Immediate Relatives of a KC have different Ontology Labels than the Ontology Label of the KC, and the Immediate Relatives of any Canonical Tree T have different Ontology Labels than the Ontology Label of T.

Proof:

Consider Figure 4.4, which shows a "paper" and an "article" Canonical Trees. They both have the same Ontology Label "publication". The two Canonical Trees contain nodes "title" and "year", which are characteristics of the "publication" supertype. However, each of them has its own specific node: "conference" in the "paper" and "journal" in the "article" Canonical Tree. Consider that Canonical Tree "paper" is KC and let N_k denote the node containing the keyword. Below are all possible query scenarios that involve the two Canonical Trees:

Scenario 1: N_k is "title": If the IAN is "year", then intuitively this year is the one contained in the KC and not the one contained in the "article" Canonical Tree. If IAN is "conference", then obviously it is the one contained in the KC.

Scenario 2: N_k is "year": Similar to scenario 1.

Scenario 3: N_k is "conference": If the IAN is "title" or "year", then intuitively they are the ones contained in the KC and not the ones contained in the "article" Canonical Tree. If, however, the IAN is "journal", then obviously, the query is meaningless and unintuitive, since the user wants to know a journal's name by providing a conference's

name. The query would be unintuitive even if both the "paper" and "article" are authored by the same author.

As the example scenarios show, the IAN cannot be contained in the "article" Canonical Tree if the KC is the "paper" Canonical Tree. That is, if the IAN of an intuitive and meaningful query is contained in Canonical Tree T, then either $OL_T \neq OL_{KC}$ or T is the KC. In other words, the Immediate Relatives of a KC have different Ontology Labels than the Ontology Label of the KC. The reason is that Canonical Trees, whose Ontology Labels are the same have common entity characteristics (because they capture information about real-world entities sharing the same type) and therefore they act as rivals and do not participate with each other in Immediate Relative relationships. They can, however, participate with each other in a relationship by being Immediate Relatives to another Canonical Tree. As an example, Canonical Trees T_4 and T_8 in Figure 4.3 have the same Ontology Label "person" and are both Immediate Relatives to Canonical Tree T_2 (the publication contained in T_2 is authored by the authors contained in T_4 and T_8). Therefore, in the answer for the query Q(XML), where T_2 is the KC, both T_4 and T_8 will be included as part of the answer subtree.

Figure 4.4: A "paper" and an "article" Canonical Trees

Property 1:

This property is based on lemma 1. When computing IR_{KC}, we prune from the Canonical Trees graph any Canonical Tree, whose Ontology Label is the same as the Ontology Label of the KC.

Lemma 2:

The Immediate Relatives of a KC that are located in the same path should have distinct Ontology Labels. That is, if Canonical Trees T and T' are both Immediate Relatives of a KC and are located in the same path from the KC, then $OL_T \neq OL_{T'}$.

Proof:

Consider that Canonical Trees T , T', and T'' are located in the same path and that T' is a descendant of T while T'' is a descendant of both T' and T . In order for T'' to be an Immediate Relative of T , then intuitively T'' has to be an Immediate Relative of T', because T' relates (connects) T'' with T . If T' and T'' have the same Ontology Label, then $T'' \notin IR_{T'}$ (according to lemma 1), and therefore $T'' \notin IR_T$. Similarly, consider Figure 4.5, which shows Canonical Trees located in the same path from the KC. The letter on top of each Canonical Tree is an OLA (see Def. 4.1 and Table 4.1) representing the Ontology Label of the C. Tree. In order for T_3 to be an Immediate Relative of the KC, it has to be an Immediate Relative of both T_2 and T_1 because they are the ones that connect and relate it with the KC. T_3 and T_1 have the same Ontology Label, therefore $T_3 \notin IR_{T1}$ (according to lemma 1). Consequently, $T_3 \notin IR_{KC}$, and $T_1, T_2 \in IR_{KC}$. Consider as another example Figure 4.3 and that the KC is C. Tree T_7. In path

T1 → T_2 → T_4 → T_7, Canonical Trees T_1 and T_4 have the same Ontology Label. Therefore, $T_1 \notin IR_{T7}$, and T_2, $T_4 \in IR_{T7}$.

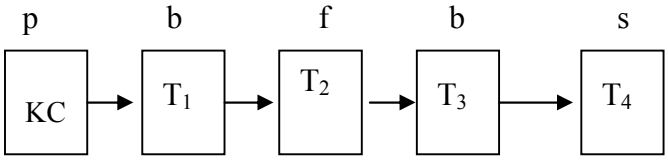

Figure 4.5: C. Trees located in the same path

Property 2:

This property is based on lemma 2. When computing IR_{KC}, we prune C. Tree T' if there is another C. Tree T'' whose Ontology Label is the same as T' and it is located between T' and the KC.

Lemma 3:

If Canonical Tree $T \notin IR_{KC}$ and Canonical Tree T' is related (connected) to the KC through T then $T' \notin IR_{KC}$.

Proof:

Every Canonical Tree T has a domain of influence. This domain covers Canonical Trees, whose *degree of relativity* to T is strong. Actually, these Canonical Trees are the Immediate Relatives of T. If by applying property 1 or 2 we have determined that Canonical Tree $T' \notin IR_T$, then the degree of relativity between T' and T is weak. Intuitively, the degree of relativity between any other C. Tree T'' and T is even weaker if T'' relates (connected) to T through T', due to the proximity factor. Consider for

example Figure 4.3 and that C. Tree T_9 is the KC. By applying property 1, $T_2 \notin IR_{T_9}$ because T_2 has the same Ontology Label as T_9. Canonical Trees T_1, T_{12}, T_3, T_4, T_5, T_6, and T_7 are connected and related to T_9 through T_2. As can be seen, the degree of relativity between each of these C. Trees and T_9 is even weaker than that between T_2 and T_9. Therefore, each of them $\notin IR_{T_9}$.

Property 3:

This property is based on lemma 3. When computing IR_{KC}, we prune from the Canonical Trees graph any Canonical Tree that is related (connected) to the KC through a Canonical Tree $T \notin IR_{KC}$.

Examples 4.1 and 4.2 below illustrate how properties 1, 2, and 3 are applied for determining IR_T. Recall Figure 4.3 for the examples.

Examp1e 4.1: Determination of IR_{T_8}:

By applying property 1, T_1 and T_4 are pruned, because their Ontology Labels are the same as the Ontology Label of T_8. By applying property 3, T_{12}, T_5, T_6, and T_7 are pruned because they relate to T_8 through either T_1 or T_4. The remaining C. Trees in the C. Trees Graph are IR_{T_8} (see Figure 4.6).

Examp1e 4.2: Determination of $IR_{T_{10}}$:

By applying property 2, T_2 is pruned, because it is located in path $T_1 \rightarrow T_2 \rightarrow T_8 \rightarrow T_9 \rightarrow T_{10}$ and its Ontology Label is the same as the Ontology Label of T_9, which is closer to T_{10}. By applying property 3, T_1, T_{12}, T_3, T_4, T_5, T_6, and T_7 are pruned, because they relate to T_{10} through T_2. The remaining C. Trees in the C. Trees Graph are $IR_{T_{10}}$ (see Figure 4.7).

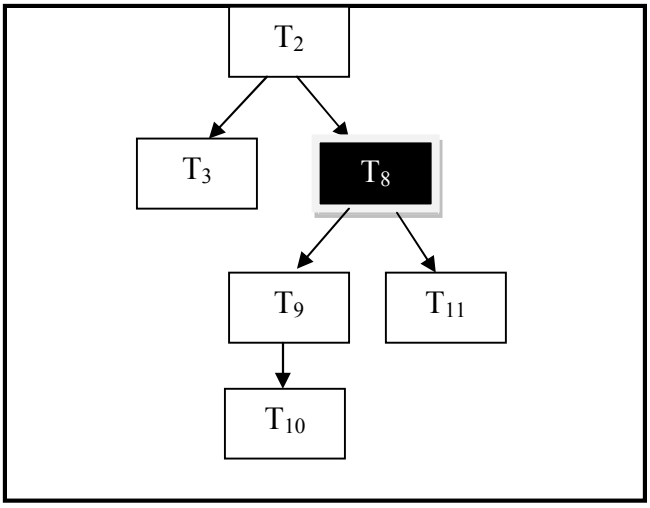

Figure 4.6: IR_{T8}

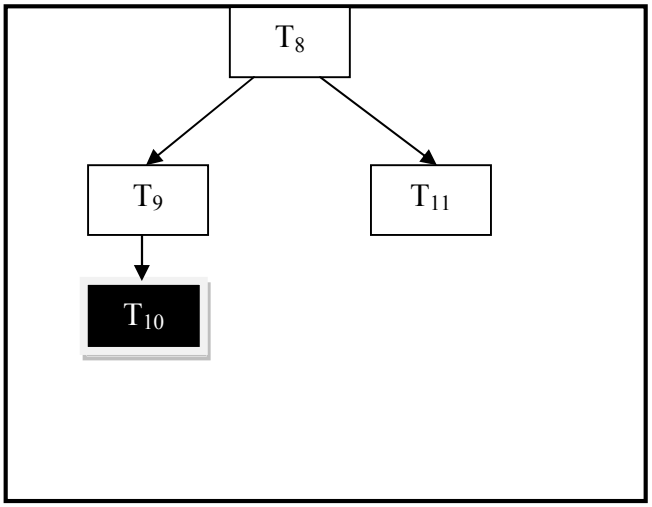

Figure 4.7: IR_{T10}

Let us consider the following 2 queries – both of type A.

Query1:

Q("Joe Smith", title)

Here, the search term element is "name = Joe Smith" and the return element is "title". We need the title of the publication authored by "Joe Smith". Since the search term element is present in the canonical tree T_8 (see Figure 4.3), T_8 is the STC. The answer would be either in T_8 or in the canonical trees comprising IR_{T8}. For calculation of IR_{T8}, refer Example 4.1 and Figure 4.6. According to Figure 4.3, the nodes containing the "title" element which are also part of IR_{T8} are T_2 and T_9. T_2 represents the title of the publication which was co-authored by "Joe Smith" and T_9 represents the title of the publication which was authored by "Joe Smith". Both these nodes represent the correct answer.

Query2:

Q("VLDB", title)

Here, the search term element is "name = VLDB" and the return element is "title". We need the title of the publication in the proceedings "VLDB". Since the search term element is present in the canonical tree T_{10} (see Figure 4.3), T_{10} is the STC. The answer would be either in T_{10} or in the canonical trees comprising IR_{T10}. For calculation of IR_{T10}, refer Example 4.2 and Figure 4.7. According to Figure 4.3, the node containing the "title" element which is also part of IR_{T10} is T_9. T_9 represents the title of the publication in the proceedingsSponsor "VLDB". T_9 represents the correct answer.

Query type B concerns the types of queries where the user knows some information about something and wants additional information. In this type of query, the search term element and return element have the same labels. For example a user who knows that "XML" is a title of one of the books authored by "John" and wants to know the titles of the other books and articles authored by him could submit a loosely structured query consisting of the search term "title = XML" and return element "title". When answering this query, we ONLY look for Canonical Trees, whose Ontology Labels are "publication". So, when answering query type B, we look only for Canonical Trees, whose Ontology Labels are the same as the Ontology Label of the Canonical Tree that contains the search term.

Property 4:

When a query's search term element and return element have the same label, we get the answer return element node(s) as follows. We look at the Canonical Trees graph (e.g. Figure 4.3) and the set of IRs. In the Canonical Trees graph and starting from the Canonical Tree T_i *that contains the search term node we search ascending and descending* T_i *for the closest Canonical Tree, whose IR contains* T_i *and also contains at least one more Canonical Tree, whose Ontology Label is the same as* T_i*. We call this Canonical Tree the pivoting entity and its IR contains the query's answer return element node.*

Example 4.3: **for** $d **in** doc("student.xml")

where $d//name = "Tom Wilson "

return $d//name.

The query asks for the names of the *other* authors of the "publication" authored by Tom Wilson (see Figure 4.3). The keyword "Tom Wilson" is contained in T_1. The closest Canonical Tree to T_1, whose IR contains T_1 and also contains another Canonical Tree(s), whose Ontology Label(s) is/are the same as T_1, is Canonical Tree T_2. So, the pivoting entity is T_2 "publication". Therefore, we use IR_{T2} (see Figure 4.8). The answer is nodes 10 and 21 contained in T_4 and T_8 respectively.

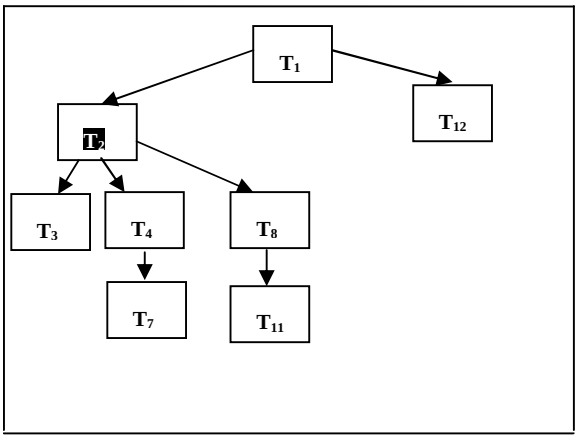

Figure 4.8: IR_{T2}

CHAPTER 5

CXLENGINE - A COMPREHENSIVE XML LOOSELY STRUCTURED SEARCH ENGINE

5.1 Motivation

The main objective of this thesis is to make the OOXSearch a fault free framework. In an XML scenario where the parent and the child have the same Ontology Label and are having leaf children nodes, the OOXSearch returns erroneous results. In Figure 5.1 for example, Canonical Trees T_1 and T_8 satisfy this condition. Similarly, T_8 and T_9 also satisfy the above mentioned condition. To illustrate the problem in OOXSearch, consider a query, whose KC is Canonical Tree T_1. According to property 1, T_8 and T_9 are not Immediate Relatives of T_1 and should be pruned. Intuitively, T_8 is an Immediate Relative of T_1 and shouldn't be pruned. Consider the query "what is the name of the GRA, who is working with James King". The example shows that T_8 (which contains the name of the GRA) is an Immediate Relative of T_1 (which contains the keyword James King). If we use OOXSearch, T_8 would get pruned and we would not get the correct result. We can solve this problem by changing the Ontology Label of the child Canonical Tree. In this thesis, we propose an extension to the OOXSearch engine that solves the above mentioned problem.

5.2 Algorithm

We constructed algorithm ComputeIRs for computing IR_T of Canonical Tree T. The algorithm first calls subroutine Rename (line 2), which proceeds as follows. If a

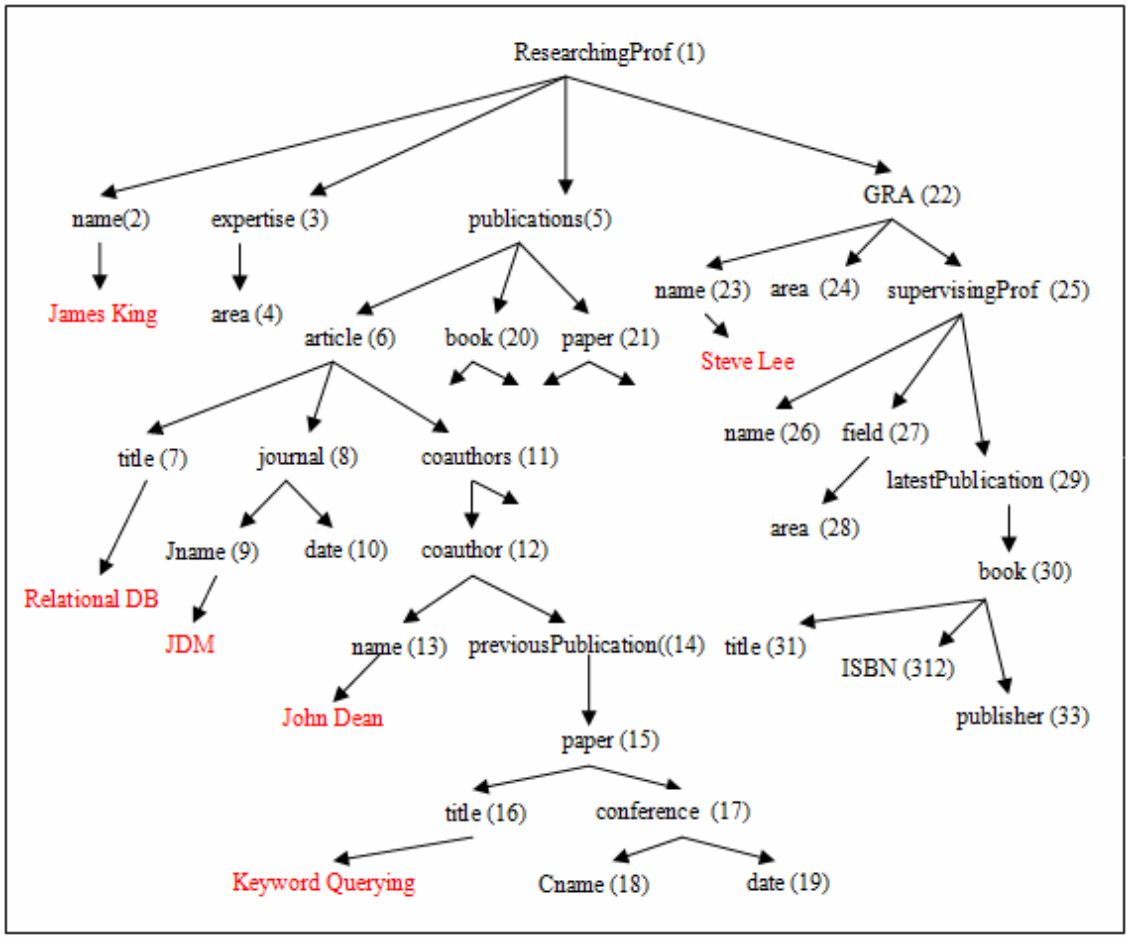

Figure 5.1: Graduate School Publications XML tree (publication.xml)

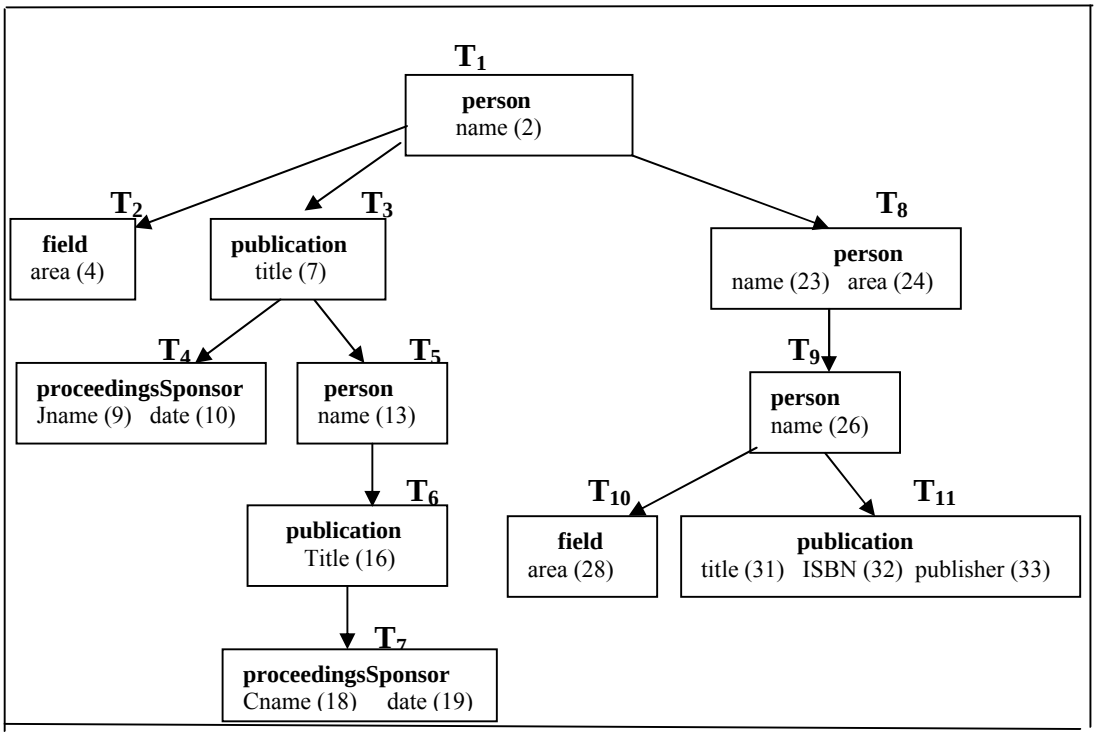

Figure 5.2: Canonical Tree Graph for the XML tree in Figure 5.1

Canonical Tree T' is adjacent to Canonical Tree T in the Canonical Trees graph (T' is either a child or the parent of Canonical Tree T) and if it's Ontology Label is the same as the Ontology Label of T(line 2), then the subroutine would relabel the Ontology Label of T' by attaching underscore followed by a digit to the Ontology Label of T' ((line 5), and then stores the new label in a hash table called $OL_{T_i}^{TBL}$ (line 6). Otherwise, if T' does not satisfy the condition of line 2, the subroutine would store the original Ontology Label of T' in table $OL_{T_i}^{TBL}$ (line 7). After the relabeling, algorithm ComputeIRs then calls subroutine DetermineIRs (line 5) to compute IR_T. To compute IR_T, instead of examining each C. Tree in the graph to determine if it satisfies one of the three properties, subroutine DetermineIRs examines only the C. Trees that are *adjacent* to any C. Tree T' $\in$ IR_T. If the subroutine determines that C. Tree T' $\in$ IR_T, it will then examine its adjacent C. Trees. However, if the subroutine determines that T' $\notin$ IR_T, it will not examine any C. Tree T'' that is connected to T through T', because T'' is known to be not an Immediate Relative of T, according to property 3. Set $s_{KC}^{T'}$ (line 4) stores the Ontology Label of each C. Tree located between C. Tree T' and the KC. In line 2, if the Ontology Label of T' is not the same as that of the KC (T' is not satisfying property 1) or it is not included in set $s_{KC}^{T'}$ (T' is not satisfying property 2), then T' $\in$ IR_T (line 3) and we recursively examine the adjacent C. Trees of T' (line 5). Otherwise, if T' $\notin$ IR_T, all C. Tree connected with T through T' will not be examined. The time complexity of the algorithm is $O(\sum_{i=1}^{|T|} |IR_{T_i}|)$

ComputeIR
{

1. count ← 1 /* count is an increment variable*/

2. Rename (T, count) /* call subroutine *Rename* */

3. S_{KC}^{T} ← null /* Set S_{KC}^{T} contains $OL_{T'}$ between T and KC */

4. IR_{KC} ← null /* IR_{KC} is a set containing the IR of a KC */

5. DetermineIRs (T) } /* call subroutine DetermineIRs */

Figure 5.3: Algorithm ComputeIR

Rename (T **, count) {**

1. **for** each Canonical Tree T' {
   /* Check if T' is a child/parent of T and if it has the same OL as T */

   2. **if** (T' ϵ Adj[T] & $OL_{T'} = OL_T$)

      3. **then** {

      4. count = count + 1

      5. tempOL ← $OL_{T'}$ + '_' + count /*Change the Ontology Label of the child C. Tree by attaching "_digit*/

      6. Add tempOL to table OL_T^{TBL} as $OL_{T'}$

      }

   7. **else** add $OL_{T'}$ (the original OL of T') to table OL_T^{TBL}

   }

}

Figure 5.4: Subroutine Rename

```
DetermineIRs {                    /*Compute Immediate Relatives of T */
1. for each Canonical Tree T' ∈ Adj[T]      /*T' is adjacent to T */
2.   if OL_T' ≠ OL_KC && OL_T' ∉ S_KC^T'    /*T' doesn't satisfy property 1 & 2*/
3.       then { IR_KC = IR_KC + T'          /*T' ∈ IR_KC */
4.              S_KC^T' ← S_KC^T + OL_T'
5.              DetermineIRs (T')
               }
        }
```

Figure 5.5: Subroutine DetermineIRs

Example 5.1: Let's compute IR_{T1} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.1.

Table 5.1: Hash Table $OL_{T_1}^{TBL}$

T_1	person
T_2	field
T_3	publication
T_4	proceedingsSponsor
T_5	person
T_6	publication
T_7	proceedingsSponsor
T_8	**person_1**
T_9	person
T_{10}	field
T_{11}	publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T1} as shown in Figure 5.6.

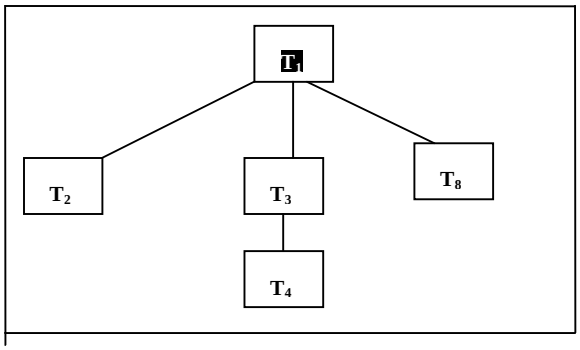

Figure 5.6: IR_{T1}

Example 5.2:

for $d **in** doc("publication.xml")

where $d//name = "James King"

return $d//name

The query asks for the name of the GRA, who works with professor "James King". Since the keyword "James King" is contained in Canonical Tree T_1, we use IR_{T1} (recall Figure 5.6). The answer return element node "name" is node 23, which is contained in Canonical Tree T_8.

We have shown the computation of Immediate Relative for the canonical trees in Figure 4.3. We have also presented the hash table $OL_{T_i}^{TBL}$ for the canonical trees in Figure 4.3.

Example 5.3: Let's compute IR_{T2} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.2.

Table 5.2: Hash Table OL_{T2}^{TBL}

T_1	person
T_2	field
T_3	publication
T_4	proceedingsSponsor
T_5	person
T_6	publication
T_7	proceedingsSponsor
T_8	person
T_9	person
T_{10}	field
T_{11}	publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T2} as shown in Figure 5.7.

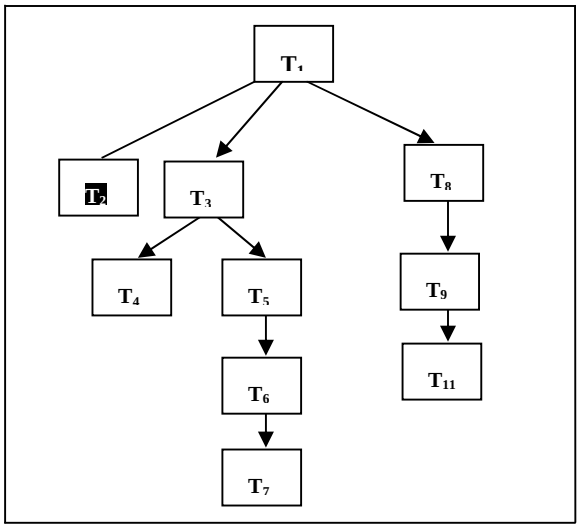

Figure 5.7: IR_{T2}

Example 5.4: Let's compute IR_{T3} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.3.

Table 5.3: Hash Table OL_{T3}^{TBL}

T_1	Person
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	Person
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T3} as shown in Figure 5.8.

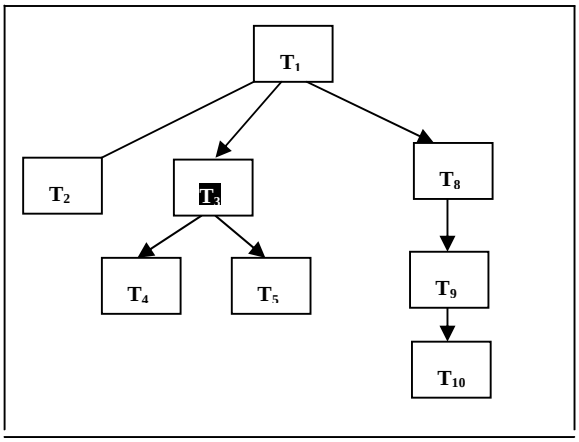

Figure 5.8: IR_{T3}

Example 5.5: Let's compute IR_{T4} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.4.

Table 5.4: Hash Table OL_{T4}^{TBL}

T_1	Person
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	Person
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T4} as shown in Figure 5.9.

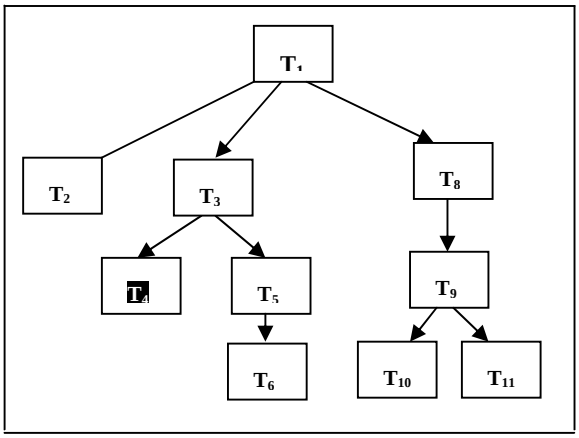

Figure 5.9: IR_{T4}

Example 5.6: Let's compute IR_{T5} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.5.

Table 5.5: Hash Table OL_{T5}^{TBL}

T_1	Person
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	Person
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T5} as shown in Figure 5.10.

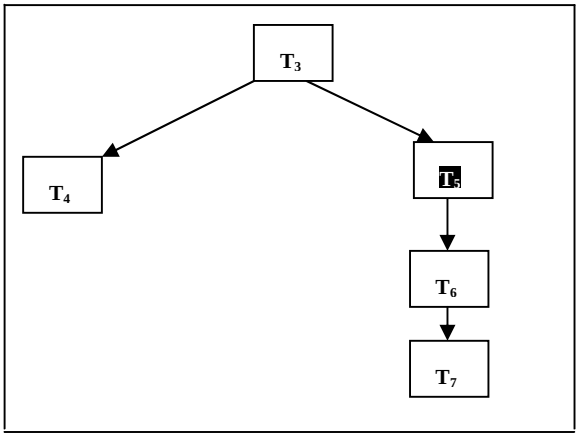

Figure 5.10: IR_{T5}

Example 5.7: Let's compute IR_{T6} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.6.

Table 5.6: Hash Table OL_{T6}^{TBL}

`

T_1	Person
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	Person
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T6} as shown in Figure 5.11.

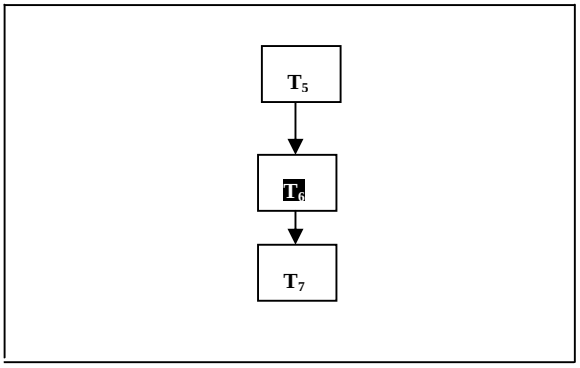

Figure 5.11: IR_{T6}

Example 5.8: Let's compute IR_{T7} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.7.

Table 5.7: Hash Table OL_{T7}^{TBL}

T_1	Person
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	Person
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T7} as shown in Figure 5.12.

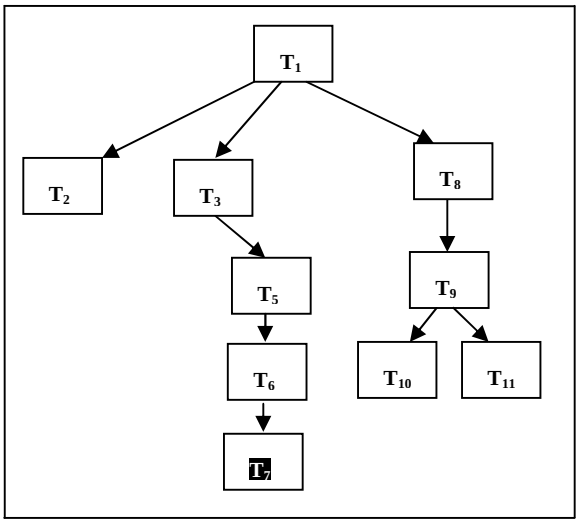

Figure 5.12: IR_{T7}

Example 5.9: Let's compute IR_{T8} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.8.

Table 5.8: Hash Table OL_{T8}^{TBL}

T_1	**person_8**
T_2	Field
T_3	Publication
T_4	proceedingsSponsor
T_5	Person
T_6	Publication
T_7	proceedingsSponsor
T_8	Person
T_9	**person_8**
T_{10}	Field
T_{11}	Publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T8} as shown in Figure 5.13.

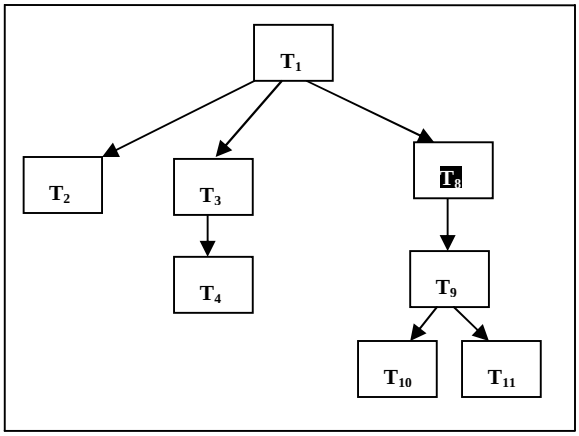

Figure 5.13: IR_{T8}

Example 5.10: Let's compute IR_{T9} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.9.

Table 5.9: Hash Table OL_{T9}^{TBL}

T_1	person
T_2	field
T_3	publication
T_4	proceedingsSponsor
T_5	person
T_6	publication
T_7	proceedingsSponsor
T_8	**person_9**
T_9	person
T_{10}	field
T_{11}	publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T9} as shown in Figure 5.14.

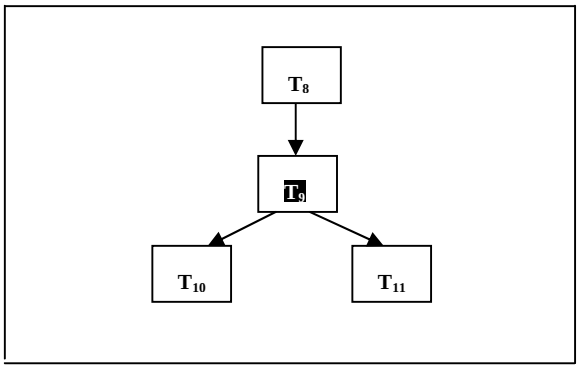

Figure 5.14: IR_{T9}

Example 5.11: Let's compute IR_{T10} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.10.

Table 5.10: Hash Table OL_{T10}^{TBL}

T_1	person
T_2	field
T_3	publication
T_4	proceedingsSponsor
T_5	person
T_6	publication
T_7	proceedingsSponsor
T_8	person
T_9	person
T_{10}	field
T_{11}	publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T10} as shown in Figure 5.15.

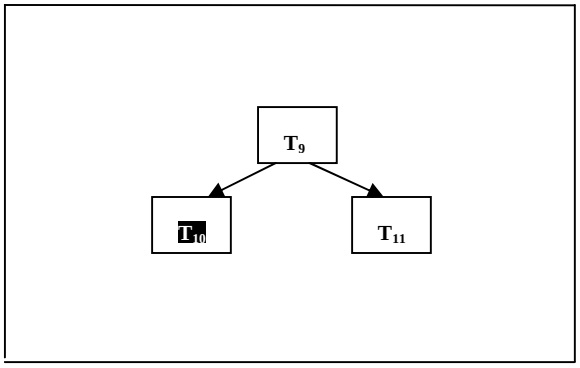

Figure 5.15: IR_{T10}

Example 5.12: Let's compute IR_{T11} using algorithm ComputeIRs. Line 2 calls subroutine Rename. This subroutine will return the hash table shown in Table 5.11.

Table 5.11: Hash Table OL_{T11}^{TBL}

T_1	Person
T_2	field
T_3	publication
T_4	proceedingsSponsor
T_5	person
T_6	publication
T_7	proceedingsSponsor
T_8	person
T_9	person
T_{10}	field
T_{11}	publication

The algorithm would then call subroutine DetermineIRs (line 5). This subroutine would return IR_{T11} as shown in Figure 5.16.

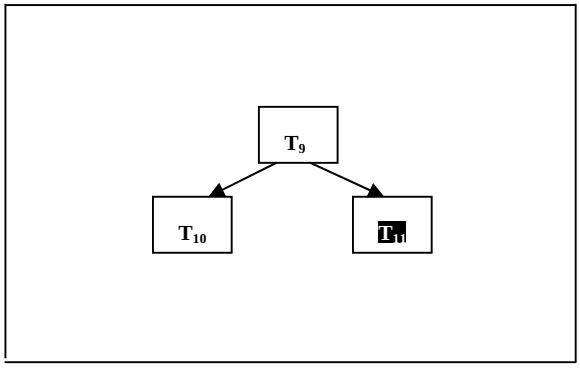

Figure 5.16: IR_{T11}

5.3 System Implementation

Figure 5.17 shows the system architecture. The OntologyBuilder uses an ontology editor tool to create ontologies and populates them with instances. We used Protégé ontology editor [14] in the implementation. [13] lists these tools. The XML schema describing the structure of the XML doc is input to the OntologyBuilder, which outputs to the GraphBuilder the list of Ontology Labels corresponding to the interior nodes in the XML schema. Using the input XML schema and the list of Ontology Labels, the GraphBuilder creates a Canonical Trees graph. The Re-labeler uses algorithm Rename To re-label the Ontology Labels of the Canonical Trees graph, as described in section 5.2. For each Canonical Tree T_i , its $OL_{T_i}^{TBL}$ are stored in a hash table called $OL_{T_i}^{TBL}$. Using this hash table and the Canonical Trees graph, the IRdeterminer uses algorithm DetermineIRs to compute for each Canonical Tree T_i its $OL_{T_i}^{TBL}$, which is stored in a hash table called IR_TBL. Using the input XML doc and the Canonical Trees Graph, the IndexBuilder stores in a table called Keyword Table for each keyword the Canonical Trees containing it. This table is saved in a disk. After the Query Engine receives a Loosely Structured query, it uses the Keyword Table to locate the KC(s), and uses the IR_TBL to determine the IAN(s). The data contained in the IAN(s) is extracted using XQuery Engine.

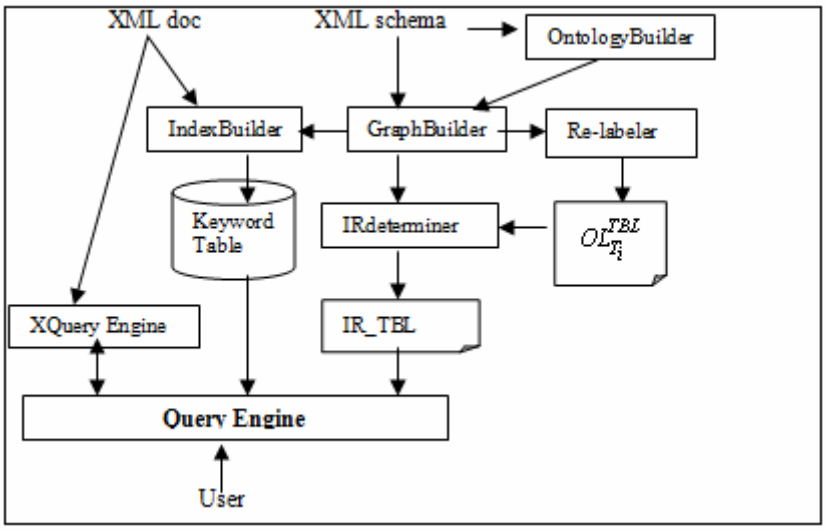

Figure 5.17: CXLEngine system architecture

Due to the resemblance between an object and a canonical tree, OO programming is the most efficient way to extract answer nodes from the relevant canonical trees. OO Programming can also be used to extract the instance values from the data nodes. The relationship between the STC and its IR can be considered as analogous to the relationship between objects in an OO environment. Every canonical tree is associated with a class. The canonical trees whose ontology labels are the same behave as subclasses. These sub classes inherit the common properties and attributes from the superclass. This superclass has the name of the ontology labels of the subclasses. If we consider Figure 5.1 and Figure 5.2, the “person” and “publication” classes can be considered as superclasses and the “researchingProf”, ”GRA”, “book” and “article” classes can be considered as the subclasses that inherit from the superclasses.

5.4 Experimental Results

The implementation of CXLEngine was done in JAVA. The implementation of [15] was modified and expanded for the implementation of CXLEngine. The experiments were carried out using a AMD Athlon XP 1800+ processor, with a CPU of 1.53 GHz and 736 MB of RAM. Windows XP Operating system was used. CXLEngine was evaluated by comparing it with [15].

5.4.1. Recall and Precision Evaluation

Let us look at the definitions of recall and precision that were presented earlier. Recall is the ratio of the number of relevant records retrieved to the total number of relevant records in the database, and precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved.

Data models from two different sources were used. The first one is XMark benchmark [20]. XMark provides 20 queries written in schema aware XQuery accompanied by a 100 MB XML document. The second is XML Query Use Cases provided by W3C [19]. The XML documents were generated using [16]. Figure 5.18 and Figure 5.19 show the average recall and precision of OOXSearch and CXLEngine on the test data. As can be seen from the graphs, the performance of CXLEngine is better than that of OOXSearch. This is because of the relabeling techniques that were incorporated in CXLEngine. OOXSearch returns wrong answers for the cases where the parent and child have the same ontology label and are having leaf children nodes. The relabeling algorithm proposed in this thesis (Figure 5.4) overcomes this problem and thus, the recall and precision of CXLEngine is better than that of OOXSearch.

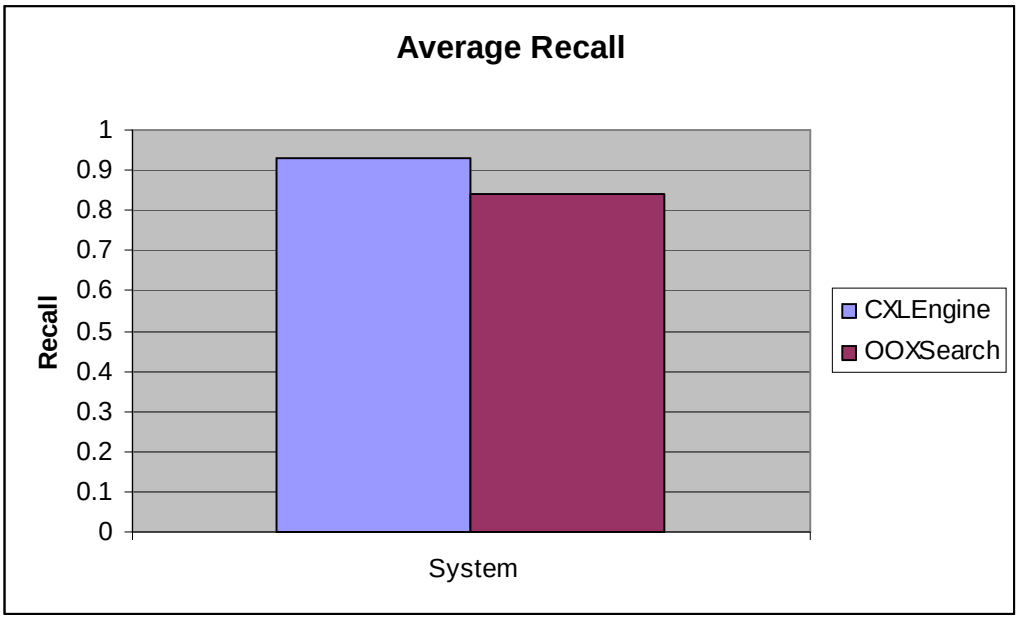

Figure 5.18: Average Recall

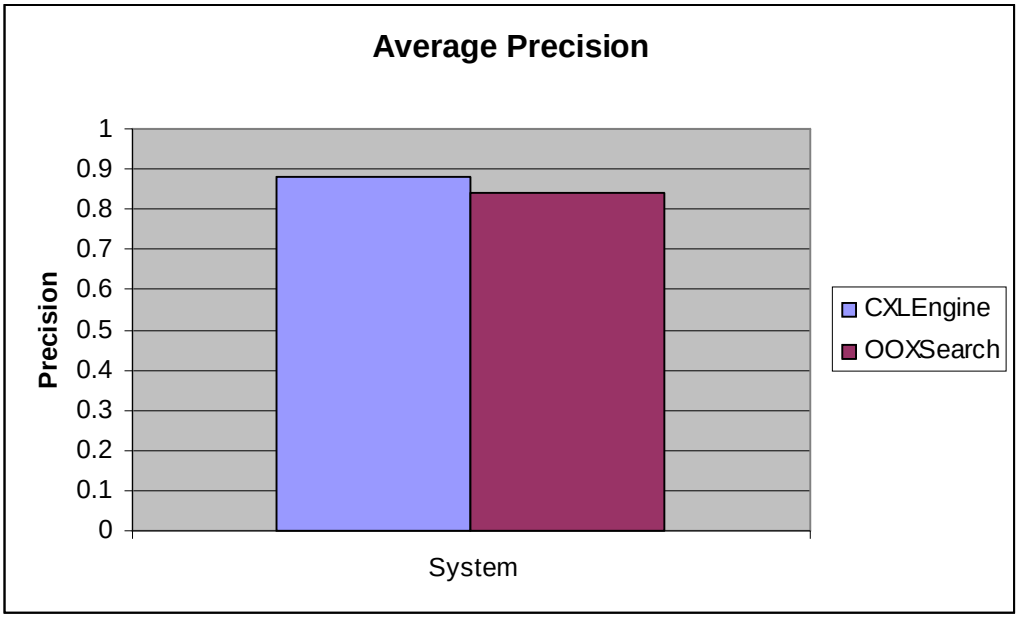

Figure 5.19: Average Precision

5.4.2. Search Performance Evaluation

To evaluate the query execution times of CXLEngine under different document sizes, we ran queries using different doc sizes (150, 200, 250, and 300 MB) of XMark [20], and compared it with OOXSearch. We ran 15 queries and obtained the average query execution time. Figure 5.20 shows the result. As can be seen from the figure, the query execution time of CXLEngine is higher than that of OOXSearch. This can be attributed to the fact that CXLEngine has the extra overhead of the relabeling algorithm. (Figure 5.4). As can be seen from the graph, the execution times of CXLEngine range around 120% of the execution times of OOXSearch.

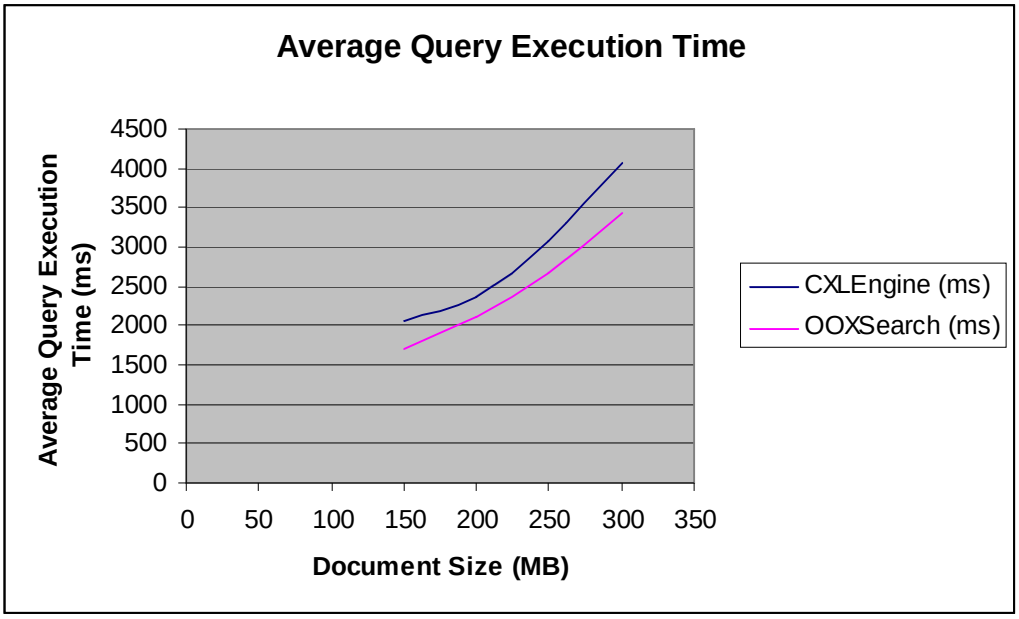

Figure 5.20: Average Query Execution Time

CHAPTER 6

CONCLUSION

This thesis is an extension of the framework proposed in [15]. The OOXSearch engine takes into consideration the context of nodes (their parents). The main contribution of [15] was the employing of semantic relationships between nodes based on their contexts and also the use of OO programming to extract data from those nodes. The proposed framework is efficient, robust, and works in heterogeneous data models. OOXSearch returned faulty recall and precision for one of the data models where the parent and the child belong to the same type and are having leaf children nodes. The main contribution of this thesis is that it makes the OOXSearch a fully fault free framework by proposing a solution for the above mentioned problem.

CHAPTER 7

FUTURE WORK

The CXLEngine accepts queries that are loosely structured where the user knows the search term label and the return term label in addition to the keyword. The user need not know the structure of the underlying data. In our future work, CXLEngine can be expanded to be a search engine that accepts the following three kinds of queries

- *Fully Structured*

  User is aware of the entire schema

- *Loosely Structured*

  User need not be aware of the entire schema and is required to know only the label of the search term element and the return element.

- *Keyword*

  User need not know the schema or any complex query language. He/she just needs to know the keyword they want to search.

REFERENCES

[1] Amer-Yahia, S., Deutsch, *Flexible and Efficient XML Search with Complex Full-Text Predicates*. In SIGMOD 06.

[2] Florescu, D., Manolescu, I. *Integrating Keyword Search into XML Query Processing*. Computer Networks, 33:119-135, 2000.

[3] Agrawal, C., Das, G. (2002). *DBXplorer: a System for Keyword-Based Search Over Relational Databases*. In ICDE 02.

[4] Aditya, B. and Sudarshan, S. (2002). *BANKS: Browsing and Keyword Searching in Relational Databases*. In VLDB 02.

[5] Balmin, B., Hristidis, V., and Koudas, N. (2003). *A System for Keyword Proximity Search on XML Databases.* In VLDB 2003.

[6] Balmin, A., Hristidis, V., and Papakonstantinon, Y. (2003). *Keyword Proximity Search on XML Graphs In* ICDE 03.

[7] Balmin, B., Hristidis, V., *ObjectRank: Authority-Based Keyword Search in Databases.* In VLDB 04.

[8] Botev, C., Guo, L., Shao, F. (2003*). XRANK: Ranked Keyword Search over XML Docs*. In SIGMOD 03.

[9] Cohen, S., Mamou, J. and Sagiv, Y. (2003). *XSEarch: A Semantic Search Engine for XML*. In VLDB 03.

[10] Cohen, S., Kanza Y. *Interconnection Semantics for Keyword Search in XML*. In

CIKM 05.

[11] Hristidis, V., Papakonstantinou, Y., *DISCOVER: Keyword search in Relational Databases*. In VLDB 02.

[12] Jagadish, H. and Li, Y., *Schema-Free XQuery.* In VLDB 04.

[13] List of Ontology editor tools: http://www.xml.com/2002/11/06/Ontology_Editor_Survey.html

[14] Protégé ontology editor: http://protege.stanford.edu/

[15] Kamal Taha, Ramez Elmasri, *OOXSearch: A Search Engine for Answering Loosely Structured XML Queries Using OO Programming.* In Proceedings of the 24th British National Conference on Databases (BNCOD 07), published in LNCS 4587.

[16] ToXgene, a template-based generator for large XML docs. http://www.cs.toronto.edu/tox/toxgene/

[17] TIMBER: http://www.eecs.umich.edu/db/timber/

[18] Xu, Y. and Papakonstantinou, Y., *Efficient Keyword Search for Smallest LCAs in XML Databases.* In SIGMOD 05.

[19] XML Query Use Cases, W3C Working Draft 2006. Available at http://www.w3.org/TR/2006/WD-xquery-use-cases-20060608/

[20] XMark — An XML Benchmark Project. Available at http://monetdb.cwi.nl/xml/downloads.html

[21] XML Validation Benchmark, Sarvega, Available at: http://www.sarvega.com/xml-validation-benchmark.html

[22] *XQEngine version 0.69*, downloaded from http://sourceforge.net/projects/xqengine

[23] V. Hristidis, L. Gravano, Y.Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. VLDB, 2003

BIOGRAPHICAL INFORMATION

Indhu Krishna Sivaramakrishnan is a Masters Student in the Department of Computer Science at the University of Texas, Arlington. She received her Bachelor of Engineering Degree from Madurai Kamaraj University, Madurai, India. She worked as a Programmer Analyst in Cognizant Technology Solutions in India prior to her Masters. Her research interests include XML Keyword Search and Algorithms.