A Framework for Enhancing Non Context-Driven XML Search

Kamal Taha, Ramez Elmasri
Department of Computer Science and Engineering
The University of Texas at Arlington
701 S. Nedderman Drive, Arlington, Texas 76019, USA
{kamal.taha,elmasri}@uta.edu

ABSTRACT: A data element specifies one of the characteristics of its parent element. Therefore, the context of a data element is determined by its parent. Non context driven search engines build relationships between data nodes based solely on their labels and proximity to one another while overlooking their contexts. Therefore, they may return faulty answers. This paper investigates the pitfalls and limitations of non context-driven XML search engines, which caused by overlooking elements’ contexts. We propose a generic context-driven search framework, which could be used as a layer on top of the frameworks of non context-driven search systems.

Categories and Subject Descriptors
[Information Storage and Retrieval]: Information Search and Retrieval - Information filtering; Selection process; Retrieval models

General Terms: XML, XML Search engine, Keyword search, Loosely structured-based querying

Keywords: XML Search engine, XML, Keyword search, Context-driven search system

Received: 11 July 2009; Revised 19 October 2009; Accepted 4 November 2009

1. Introduction

There has been extensive research in XML keyword-based and loosely structured querying. A keyword query is composed of a set of keywords relevant to the information that the user is looking for. Loosely structured querying allows combining some structural constraints within a keyword query, by specifying the context where a search term should appear (combining keywords and element names). The popularity of keyword-based and loosely structured-based querying stems from the fact that they are simple, user friendly, and do not require the user to learn a query language. A number of studies [1, 2, 3] proposed frameworks for answering XML keyword and loosely structured queries. Some frameworks work well for certain types of XML data models and fail in others. The Computation of the Lowest Common Ancestors (LCAs) of elements containing keywords is the common denominator among the proposed search engines. 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. The context of a data element is determined by its parent element, because a data element specifies one of the characteristics of its parent. Building relationships between data elements without consideration of their parents may result in relationships that are semantically disconnected. Non context-driven search engines [1, 2, 3] 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. If for example a data element is labeled “title”, one cannot determine whether it refers to a book title or a job title without referring to its parent. Consider as another example that an XML document containing two elements labeled “name”, one of them refers to the name of a student, while the other refers to the student’s school name. Building a relationship between the two “name” elements without considering of their parents may lead to the incorrect conclusion that the two elements belong to the same type.

This paper discusses the pitfalls and limitations of non context-driven XML search engines, which caused by overlooking nodes’ contexts. By taking [1, 2, 3] as sample of non context driven XML search systems, the paper demonstrates the pitfalls and limitations of non context-driven approach. These problems could be avoided if the search engines employ semantic mechanisms that consider the contexts of elements. In this approach, the relationships between data elements are determined based on the type of their parents. We describe this approach in the paper and present a generic context-driven framework.

The rest of this paper is organized as follows. In section 2, we describe the limitations of context-driven search systems using sample of queries and three non context-driven search engines. Section 3 describes a generic context-driven framework. We present the conclusions in section 4.

2. Limitation of non Context-Driven XML Search Systems

Non context-driven search engines may return a faulty answer for a query that meets one of the following criteria:

- The search term element or the return element of the loosely structured query has the same label as one of the nodes in the XML tree that have the same label and type.
- The query is submitted against an XML document, whose XML tree contains: (1) a set S of nodes having ancestor-descendant relationships and containing all the query’s keywords, and (2) a node $e \in S$ containing one of the query’s keywords.
- The query is submitted against an XML document, whose XML tree contains nodes having different labels but the same type.
- The query is submitted against an XML document, whose XML tree contains nodes having the same label but different types.
- The loosely structured query’s search term and return elements have the same label.

Let us take [1, 2, 3] as sample of non context driven XML search systems. Sections 2.1, 2.2, and 2.3 describe the techniques used by each of the three systems as well as the pitfalls and limitations caused by overlooking nodes’ contexts. These pitfalls
are demonstrated by running query samples against the data shown in Figures 1, 2, and 3. Below are brief descriptions of the contents of Figures 1 and 3.

- Figure 1: The paper (node 4) was authored by: (1) a student (node 1), (2) a contributing student (node 9), and (3) 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.

- Figure 3: The customer (node 1) ordered a book (node 10) and had previously ordered a magazine (node 35). The author (node 13) authored the book (node 10) as well as another publication (node 17).

2.1 Schema-Free XQuery System

Schema-Free XQuery [2] uses an algorithm called MLCAS for computing the Meaningful Lowest Common Ancestor (MLCA) of nodes containing keywords. Nodes a and b are considered meaningfully related and their Lowest Common Ancestor (LCA) node c is considered the MLCA of a and b, if c is not 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 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. Algorithm MLCAS uses a stack, with the head of each stack node being a descendant of the stack node below it. The basic idea is to perform one single merge pass over the nodes, in the order of their traversal in the XML tree using Depth First Search, and conceptually merge them into rooted trees containing MLCA. Within each such tree, the root is a MLCA, and the leaf level contains nodes from different MLCA sharing the same root. The time complexity of the algorithm is \(O(h \sum_{i=1}^{m} n_i)\) where h is the height of the XML tree, m is the number of input nodes, and \(n_i\) is the number of nodes having the same label as node i. A loosely structured query in [2] has the same form of XQuery [4], but only the descendant operator (//) is used to indicate that the terms in the query are descendants of the root element. So, the format of the query is as follows:

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

The following sample of queries demonstrates the problems of the techniques employed by [2]:

- Consider Figure 1 and the query:

```xml
for $d$ in doc("doc name")
where $d$/name = "Tom Wilson"
return $d$/name
```

**Figure 1. A graduate school's authors and coauthors bibliography XML tree**

**Figure 2. Fragments of XML does taken from XML Validation Benchmark [4, 5]**
The query asks for the names of the coauthors of the publication that was authored by "Tom Wilson". The correct answer is nodes 10 and 21. However, [2] would return null as the answer for the query. 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, [2] would consider node 10 is related to node 21 and not to node 2 and that node 4 is the MLCA of nodes 10 and 21. In the same token, [2] would not return node 21, because it considers it related to node 10 and not to node 2.

2.2 XSEarch System

XSEarch [1] uses an algorithm called Compute Interconnection Index, which employs dynamic programming to compute the relationships between all pairs of nodes in an XML tree. If
the relationship tree of nodes a and b (the set of nodes in the path from a to b) contains two or more nodes with the same label, then nodes a and b are unrelated; otherwise, they are related. Consider for example Figure 1 and the relationship between nodes 2 and 32. The two nodes are related, since their relationship tree (which contains nodes 2, 1, 31, 32) does not contain more than one node having the same label. On the other hand, node 2 is unrelated to node 13, because their relationship tree of nodes 2 and 19 and the relationship tree of nodes 2 and 30 do not contain two or more nodes with the same label.

Consider Figure 2-A and the query $\Omega("Tom Wilson", area)$. The query asks for the research interest area of "Tom Wilson". The correct answer is node 32. Instead of returning node 32 only, [1] would return also nodes 19 and 30, because the relationship tree of nodes 2 and 19 and the relationship tree of nodes 2 and 30 do not contain two or more nodes with the same label.

Consider Figure 2-B and the query $\Omega("Julie", name)$. The query asks for the other coauthor of the publication that was coauthored by Julie. The correct answer is node 2. But, [1] would 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").

Consider Figure 2-C and the query $\Omega(date="5/20/06", name)$. The query could be interpreted either as "what is the customer's name, who placed an order on 5/20/06" or "what is the name of the shipping company that delivered an order placed on 5/20/06". The correct answer is node 2. "David". [1] would return both nodes 2 and 5 as an one unified answer even though the two nodes refer to two entities with different types.

Consider Figure 2-D and the query $\Omega(name="Jon", address)$. The query asks for the shipping address of Jon. The correct answer is node 3. [1] would return both node 3 "address" and node 6 "address", because the relationship tree of nodes 2 and 3 and the relationship tree of nodes 2 and 6 do not contain two nodes with the same label.

Consider Figure 3 and the query $\Omega(orderNo="10024", title)$. The query asks for the title of the publication that was ordered in number "10024". Instead of returning node 11 only, XSEarch would return also nodes 36 and 18, because the relationship tree of nodes 36 and 7 and the relationship tree of nodes 18 and 7 do not contain two or more nodes having the same label. If XSEarch employs an ontological concepts, it would have discovered that the first relationship tree contains nodes 31 (previousOrder) and 6 (currentOrder) which belong to the same type, and the second relationship tree contains nodes 10 (book) and 17 (otherPublication) which also belong to the same type (publication).

On the flip side, consider Figure 3 and the query $\Omega(ISBN:"87-11-07559-7", ISBN)$. The query asks for the ISBN of the publication that had been ordered by the same customer, who ordered a publication with ISBN "87-11-07559-7". Instead of returning node 37, XSEarch would return node 19, because the relationship tree of nodes 37 and 12 contains more than one node having the same label (nodes 33 and 8 and also nodes 34 and 9). And, the relationship tree of nodes 19 and 12 does not contain more than one node having the same label.

### 2.3 XKSearch System

XKSearch [3] uses an algorithm called the Stack Algorithm to compute the Smallest Lowest Common Ancestor (SLCA) of nodes containing keywords. The algorithm is based on stack sort-merge approach. It computes the longest common prefix of each node and the node denoted by the top entry of the stack. It then pops the top entries containing Dewey components that are not part of the common prefix. If a popped entry contains all keywords, it is considered a SLCA. The SLCA is a root of a subtree, 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 1 and the query $\Omega("XML", "Julie Smith")$. Consider that node 13 contains the value "XML" instead of XQuery. Since the keyword "XML" is contained in both nodes 5 and 13, the answer subtree will be the one rooted at node 9, which contains nodes 10 and 13, and not the one rooted at node 4, which contains nodes 5 and 10. Nodes in [3] are labeled with Dewey numbers. The Dewey numbers of nodes containing keywords are stored in the stack after being merged together. [3] builds relationships between nodes containing keywords based solely on their labels and proximity to one another, while overlooking their contexts.

The following keyword-based query demonstrates the problems with the Stack Algorithm of [3]. Consider Figure 1 and the query: $\Omega("Smith", "XML", "VLDB")$. 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. As can be seen from Figure 1, there are two candidate answer subtrees. The first is rooted at node 4 and contains the three keywords in nodes 5, 10, and 15. The second is rooted at node 20 and contains the three keywords in nodes 21, 25, and 27. The first one is an incorrect answer, because the publication title "XML" (node 5) and authored by "Julie Smith" was published in an "EDBT" conference proceedings (node 7) and not in a "VLDB". Rather, the author’s publication title "XQuery" (node 13) is the one that was published in "VLDB" proceedings. The second answer subtree is a correct answer, because "Joe Smith" (node 21) authored a publication titled "XML" (node 25), which appeared in a "VLDB" journal (node 27). We show below how the Stack Algorithm of [3] answers the query by returning both the incorrect and correct answer subtrees as a result of not employing context-driven search techniques.

After labeling the nodes of the XML tree in Figure 1 with Dewey numbers, the keyword lists would be as follows. Keyword "XML" list $= [0.1.0.0, 0.1.0.3.1.1.0]$, which corresponds to nodes 5 and 25. Keyword "Smith" list $= [0.1.0.2.0, 0.1.0.3.0]$, which corresponds to nodes 10 and 21. Keyword "VLDB" list $= [0.1.0.2.1.0.0, 0.1.0.3.1.1.0]$, which corresponds to nodes 15 and 27. The third and fourth states of the Stack Algorithm of [3] are shown in Figures 4-a and 4-b, where "T" means that the node represented by the entry contains the corresponding keyword and "F" is the negation of that. The key problem of [3], which would lead it to return the faulty answer is caused by the stack in Figure 4-b, as follows. When popping the top entry in the previous stack state (the third state), [3] passes the keyword information of the popped entry to the parent entry. In the sixth stack state, [3] will return the correct answer subtree. Then, when all of the keyword lists are exhausted, line 19 of the Stack Algorithm of [3] will find that all the fields of array keywords at entry 0.1.0 contain "T" (see Figure 4-b); therefore, it will consider node 0.1.0 (node 4) a SLCA and return it as the root of an answer subtree as shown in Figure 5-a. As can be seen the semantic of the answer is incorrect, because "Julie Smith" (node 10) authored a paper titled "XML" but it did not appear in a "VLDB"
conference proceedings; rather, her publication “XQuery” is the one appeared in a “VLDB” conference proceedings. [3] could have returned only the correct answer if it employs a context-driven search technique.

A data node by itself is an entity that is semantically meaningless. If, for example, a data node is labeled “name”, we cannot determine whether it refers to a name of a student or a name of a school, unless we account for the node’s context (its parent node). The framework of FEXS accounts for the contexts of nodes as follows. It partitions XML trees to subtrees, where each subtree consists of a parent and its children data nodes. Each of these subtrees is treated as a unified entity, called a Canonical Tree (CT). A CT is a metaphor of real-world entities. In Fig. 3 for example, the parent node customer(1) along with its child data name (2) constitute a CT (see CT T1 in Fig. 6). A CT is the simplest semantically meaningful entity, and a data node by itself is not.

Two real-world entities may have different names but belong to the same type, or may have the same name but refer to two different types. To overcome that labeling ambiguity, we observe that if we cluster CTs based on the ontological concepts of their parent nodes components, we will identify a number of clusters. That is, each cluster contains CTs whose parent nodes components belong to the same ontological concept. Consider for example Fig. 1. Using this clustering scheme, we will be able to determine that the two CTs whose parent nodes components are nodes 4 “paper” and 24 “article” fall under the same cluster, since both “paper” and “article” belong to the same “publication” ontological concept. We will also be able to determine that the two data nodes labeled “name” (nodes 2 and 7) are not semantically identical (they refer to 2 different types of entities), since they belong to CTs falling under different clusters: the ontological concepts of “student” and “conference” are “person” and “publication proceedings” respectively. We use the term “OL” in the paper to refer to the ontological concept of a parent node. The OL of a CT is the OL of the parent node component of the CT. For example, the OL of CT T1 in Fig. 6 is the OL of the parent node customer (1), which is person.

A Canonical Trees Graph (CTG) is a hierarchical representation depicting the relationships between CTs. Let n and n′ be two interior nodes in an XML tree, and that they are the parent nodes components of CTs T and T′ respectively. CTs T and T′ have parent-child relationship in the CTG if there does not exist in the XML tree any node n″ on the path from n to n′ where n″ has children data nodes or attributes. Fig. 6 shows a CTG depicting the relationships between the CTs constructed from the XML tree in Fig. 3. In Fig. 3 for example, since node book(10) is a descendant of node currentOrder(6) and there is no interior node in the path from book(10) to currentOrder(6) that has children data nodes or attributes, then the CT whose parent node component is node 10 (CT T1 in Fig. 6) is a child of the CT whose parent node component is node 6 (CT T5 in Fig. 6).

3. General Context-Driven Framework

The three systems [1, 2, 3] made considerable contributions to XML keyword-based and XML loosely structured-based querying, despite their pitfalls described in section 2. Their recalls and precisions could be improved if they couple their techniques with some context-driven search technique. We propose in this section a generic context-driven search framework called FEXS (Framework for Enhancing XML Search). FEXS could be used as a layer on top of the frameworks of non-context-driven search systems.

3.1 Concepts used in FEXS

FEXS models XML documents as rooted and labeled trees. A tree i is a tuple \( i = (n, e, r, \lambda) \) where \( n \) is the set of nodes, \( e \subseteq n \times n \) is the set of edges, \( r \) is the root node of i, and \( \lambda : n \rightarrow \Sigma \) is a node labeling function where \( \Sigma \) is an alphabet of node labels. A node in a tree represents an element in an XML document. A Keyword Context (KC) is a CT containing one or more of a query’s keywords.

A data node by itself is an entity that is semantically meaningless. If for example a data node is labeled “name”, we cannot determine whether it refers to a name of a student or a name of a school, unless we account for the node’s context (its parent node). The framework of FEXS accounts for the contexts of nodes as follows. It partitions XML trees to subtrees, where each subtree consists of a parent and its children data nodes. Each of these subtrees is treated as a unified entity, called a Canonical Tree (CT). A CT is a metaphor of real-world entities. In Fig. 3 for example, the parent node customer(1) along with its child data name (2) constitute a CT (see CT T1 in Fig. 6). A CT is the simplest semantically meaningful entity, and a data node by itself is not.

Two real-world entities may have different names but belong to the same type, or may have the same name but refer to two different types. To overcome that labeling ambiguity, we observe that if we cluster CTs based on the ontological concepts of their parent nodes components, we will identify a number of clusters. That is, each cluster contains CTs whose parent nodes components belong to the same ontological concept. Consider for example Fig. 1. Using this clustering scheme, we will be able to determine that the two CTs whose parent nodes components are nodes 4 “paper” and 24 “article” fall under the same cluster, since both “paper” and “article” belong to the same “publication” ontological concept. We will also be able to determine that the two data nodes labeled “name” (nodes 2 and 7) are not semantically identical (they refer to 2 different types of entities), since they belong to CTs falling under different clusters: the ontological concepts of “student” and “conference” are “person” and “publication proceedings” respectively. We use the term “OL” in the paper to refer to the ontological concept of a parent node. The OL of a CT is the OL of the parent node component of the CT. For example, the OL of CT T1 in Fig. 6 is the OL of the parent node customer (1), which is person.

A Canonical Trees Graph (CTG) is a hierarchical representation depicting the relationships between CTs. Let n and n′ be two interior nodes in an XML tree, and that they are the parent nodes components of CTs T and T′ respectively. CTs T and T′ have parent-child relationship in the CTG if there does not exist in the XML tree any node n″ on the path from n to n′ where n″ has children data nodes or attributes. Fig. 6 shows a CTG depicting the relationships between the CTs constructed from the XML tree in Fig. 3. In Fig. 3 for example, since node book(10) is a descendant of node currentOrder(6) and there is no interior node in the path from book(10) to currentOrder(6) that has children data nodes or attributes, then the CT whose parent node component is node 10 (CT T1 in Fig. 6) is a child of the CT whose parent node component is node 6 (CT T5 in Fig. 6).

3.2 Determining Relationships in FEXS

A relationship between two CTs could be described as either immediate or extended. The Immediate Relatives of a CT T (denoted by IR T) are CTs that have strong and close relationships with T. We use the abbreviation IRN throughout the paper to denote “Intended Answer Node”. An IRN is a node containing the data that the user is looking for. If the KC of a query is CT T, each CT T′ that contains an IRN is considered an Immediate Relative of T (denoted by T′ ∈ IR T). That is, IRNs are contained in the Immediate Relatives of the KC (denoted by IRN KC). We can determine IRN KC by pruning from the CTG all CTs that are not Immediate Relatives of the KC, and the remaining ones would be IRN KC. We present below three properties that regulate the pruning process.

Property 1: When computing IRN KC, we prune from the CTG any CT, whose OL is the same as the OL of the KC.

Property 2: When computing IRN KC, we prune CT T′ from the CTG if: (1) there is another CT T′′ between T and the KC, and (2) the OL of T′′ is the same as that of T′.
Property 3: When computing $\text{IR}_{KC}$, we prune from the CTG any CT that is related (connected) to the KC through a CT $T$. $T \not\in \text{IR}_{KC}$.

The naïve approach for computing $\text{IR}_{KC}$ is to apply the three properties to all CTs in the CTG. The time complexity of this approach for computing the IR of all CTs in the graph is $O(|T|^2)$. Algorithm ComputeIR (see Fig. 7) computes $\text{IR}_{KC}$ efficiently as follows. To compute $\text{IR}_{KC}$, instead of examining each CT in the graph, we only examine the CTs that are adjacent to any CT $T'$, $T' \in \text{IR}_{KC}$. That is, if CT $T'$ is $\text{IR}_{KC}$, the algorithm will examine the CTs that are adjacent to $T'$; otherwise, it will not examine any CT $T''$ that is connected to the KC through $T'$, because $T''$ will not be an IR of the KC according to property 3. The algorithm’s time complexity is $O(\sum_i |\text{IR}_{T_i}|)$.

We now present examples to show how we can determine the Immediate Relatives of a CT using properties 1, 2, and 3.

Example 1: Let us determine $\text{IR}_{T_{12}}$ (recall Fig. 6): By applying property 1, CT $T_3$ is pruned, because its Ontology Label is the same as the Ontology Label of $T_{12}$. By applying property 3, CTs $T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}$, and $T_{11}$ are pruned because they relate to CT $T_{12}$ through the pruned CT $T_3$. The remaining CTs in the CTG are $\text{IR}_{T_{12}}$ (see Fig. 8).

Example 2: Let’s determine $\text{IR}_{T_5}$ (recall Fig. 6): By applying property 2, CT $T_5$, is pruned, because it is located in the path $T_5 \rightarrow T_4 \rightarrow T_3 \rightarrow T_1 \rightarrow T_{12}$ and its Ontology Label is the same as the Ontology Label of CT $T_3$, which is closer to CT $T_5$. By applying property 3, CTs $T_{12}, T_{13}, T_{14}, T_{15}, T_{16}$, and $T_{17}$ are pruned, because they relate to $T_5$ through the pruned $T_{12}$. By applying property 1, CTs $T_1$ and $T_{10}$ are pruned, because their Ontology Labels are the same as the Ontology Label of $T_5$. By applying property 3, CTs $T_1$ and $T_{10}$ are pruned. The remaining CTs in the CTG are $\text{IR}_{T_5}$ (see Fig. 9).

3.3. Demonstrating the Impact of FEXS on Search Effectiveness

We demonstrate in this section the impact of FEXS on search effectiveness by simulating a system called Sys_with_FEXS and comparing it with [2, 3]. Towards this, we present a sample of queries and show how Sys_with_FEXS returns correct answers for the queries and how [1, 2, 3] return faulty answers for the same queries. First, recall section 2 for the techniques used by [1, 2, 3].
Example 3:

Consider Fig. 10-A and the query $Q$ (title = "Introduction", image?). The query asks for the image presented in the section titled "Introduction" (node 3). The correct answer is node 6. But, [2] would return null. The reason is that the LCA of nodes 3 and 6 is node 2.

and node 2 is an ancestor of node 4, which is the LCA of nodes 6 and 5, and node 5 has the same label as node 3. Therefore, [2] considers node 6 is related to node 5 and not to node 3.

**Sys_with_FEXS answer:** Let $T'$ denote a CT, whose nodes components are nodes 2 and 3. Let $T'$ denote a CT, whose nodes components are nodes 4, 5, and 6. The KC is $T$. Sys_with_FEXS would consider $T' \in \text{IR}_r$, since $T'$ does not satisfy properties 1, 2, and/or 3. Therefore, it would return image node $6 \in T'$ as the answer.

Example 4:

Consider Fig. 10-B, which presents information about a conference and its colocated workshops. Nodes 4 and 7 contain the subject titles of the conference and one of its workshops. Now consider the query $Q$ (name = "ICDE", subjTitle?). The query asks for the subject title of the ICDE conference (node 2). The correct answer is node 4. But, [2] would return both nodes 4 and 7, because the LCA of each of them with node 2 is the same (node 1).

**Sys_with_FEXS answer:** Let $T$ denote a CT, whose nodes components are nodes 1, 2, 3, and 4. Let $T'$ denote a CT, whose nodes components are nodes 5, 6, and 7. The KC is $T$. Since $T$ and $T'$ have the same Ontology Label, $T' \not\in \text{IR}_r$ (recall property 1). Therefore, Sys_with_FEXS would return only the subjTitle node 4 $\in T$.

Example 5:

Consider Fig. 10-B and the query $Q$ (name = "SWOD", date?). The correct answer is null. But, [2] would return node 3, which is the date of the ICDE, and [3] would consider incorrectly that the IAN is node 3.

**Sys_with_FEXS answer:** Let $T$ denote a CT, whose nodes components are nodes 1, 2, 3, and 4. Let $T'$ denote a CT, whose nodes components are nodes 5, 6, and 7. The KC is $T$. Since $T$ and $T'$ have the same Ontology Label, $T' \not\in \text{IR}_r$, (per property 1). Therefore, Sys_with_FEXS would return null as the answer of the query.

4. Background and Conclusions

A number of studies [7, 8, 9, 10] propose modeling XML documents as graphs, and keyword queries are answered by processing the graphs based on given schemas. Others [6] use a RDBMS for answering XML keyword queries. Non context-driven XML search engines build relationships between data nodes based solely on their labels and proximity to one another while overlooking their contexts (parents). As a result, they are susceptible to returning 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 discussed and demonstrated in this paper the problems of non context-driven search engines caused by overlooking nodes’ contexts. We took [1, 2, 3] as sample and representative of non context driven XML search systems and demonstrated their pitfalls. We proposed a generic context-driven search framework, which could be used as a layer on top of the frameworks of non context-driven search engines.

References

Authors Biographies

Kamal Taha is a PhD candidate in the Department of Computer Science and Engineering at the University of Texas at Arlington, USA. He is currently working on his PhD thesis under the supervision of professor Ramez Elmasri. He received his M.S. degree in Software Engineering in 2002 from the University of St. Thomas in Minnesota, USA. He has 13 refereed publications that have appeared (or are forthcoming) in journals, conferences, and workshops proceedings, in addition to a book chapter. Kamal worked as Engineering Specialist for Seagate Technology (a leading computer disc drive manufacturer in the US) from 1995 to 2005. His research interests include XML keyword and loosely structured querying, distributed query processing and caching, recommendation systems, personalized search, and Bioinformatics Databases (Mediators, Ontologies).