Indexing XML Databases: 
Classifications, Problems 
Identification and a New Approach

Research Memorandum
CS-07-15

Mohammed Al-Badawi
Computer Science Dept
mbadawi@dcs.shef.ac.uk

Dr. Siobhán North
Computer Science Dept
s.north@dcs.shef.ac.uk

Dr. Barry Eaglestone
Information Studies Dept
b.eaglestone@sheffield.ac.uk

Date:
15th November, 2007
Executive Summary

Indexing XML databases is a critical requirement in today’s database technology because of the increase in the size and the number of users of XML databases. The literature contains an enormous number of XML indexing proposals that vary from RDBMS-based techniques to native XML approaches. In this article, these indexing techniques are reviewed for the purpose of evaluating their encoding methodologies for the hierarchal nature of XML data.

Apart from being a RDBMS-based or a native XML index, the existing XML indexes are classified into four main categories depending on the methodology they use to encode the XML’s structural relationships. The node-labelling approach is a class of XML indexes that is mainly used in XML-to-RDBMS mappings to reflect the parent-child and the ancestor-descendant relationships between the XML tree nodes. The second class of XML indexes is the path-summary approach which preserves the XML’s structural relationships by storing the all distinct XML paths so that a regular path query can be evaluated by searching the path list at a specific time. Alternatively, the sequence-based approach serializes both the XML database and the XML query and checks for the existence of the query’s sequence in the XML database sequence using sequence-based searching techniques. Similarly, the feature-based class basically encodes the XML’s structural-relationships as features in what is called a feature-matrix and use the matrix’s algebra to check for the query existence in the XML database.

To criticize the strengths and the weaknesses of the four index classes, the existing XML indexes lack efficiency in one or more of the following forms: the large index size which prevents the index residing in the computer’s main memory and therefore involves expensive I/O disk operations, the high cost in terms of the index’s construction and the query evaluation processes, and the absence and/or the high cost of the index’s update operations. Among the four index’s categories, the feature-based approaches are promising XML indexing techniques for a wider range of XML queries that are represented by regular path expressions. The approach has shown an ability to encode the XML’s structural relationships while minimizing the cost of the above deficiencies by using certain techniques from other disciplines such as the matrix/graph theories and computerised data-structures.

Based on the above fact about the feature-based approaches, this paper proposes a new feature-based XML index which use sparse matrices to encode the hierarchical structure of XML documents that are required to answer regular path queries. The index uses; sparse-matrix’s compression algorithms to minimize the index size, novel encoding methodologies for the XML’s structure to reduce the cost of constructing the index and evaluating the XML queries using the constructed index. The novel encoding methodology (used by the index) also allows systematic update operations for the index without the need to reconstruct the index when the underlying XML database changes.
Table of Contents

LIST OF FIGURES ................................................................................................................................. 7

1. INTRODUCTION .................................................................................................................................. 9

2. XML PRINCIPLES ................................................................................................................................. 10
   2.1. WHAT IS XML? ................................................................................................................................. 10
   2.1.1. XML Elements ............................................................................................................................. 11
   2.1.2. XML Attributes ............................................................................................................................ 11
   2.1.3. Comments ................................................................................................................................. 12
   2.1.4. Other XML Components ............................................................................................................ 12
   2.2. SCHEMA DEFINITION .................................................................................................................... 12
   2.2.1. DTD ........................................................................................................................................ 13
   2.2.2. XML Schema ............................................................................................................................ 13
   2.2.3. Other XML Schema Languages ................................................................................................. 14
   2.2.4. Well-Formed & Valid Documents ............................................................................................... 15
   2.3. XML DATA MODEL ........................................................................................................................ 15
       2.3.1. XML and Semi-structured Data Model ..................................................................................... 16
       2.3.2. Order .................................................................................................................................... 16
       2.3.3. XML Tree and DOM .............................................................................................................. 17
   2.4. XML QUERY LANGUAGES ............................................................................................................ 17
       2.4.1. Path Expression ...................................................................................................................... 18
       2.4.2. XPath2.0 ............................................................................................................................... 18
       2.4.3. XQuery1.0 ............................................................................................................................. 19
   2.5. SUMMARY ..................................................................................................................................... 20

3. INDEXING XML DATA .......................................................................................................................... 21
   3.1. PROCESSING DOCUMENT-CENTRIC XML ................................................................................... 21
       3.1.1. Querying Content-oriented XML: A Problem Definition ......................................................... 21
       3.1.2. Content-based Query Approaches: An Evaluation ................................................................... 22
       3.1.3. Conclusion .............................................................................................................................. 24
   3.2. TECHNIQUES FOR MEMORY-BASED XML INDEXING ................................................................. 24
       3.2.1. Structural Summaries .............................................................................................................. 24
       3.2.2. Selective XML Indexes ........................................................................................................... 25
       3.2.3. Adaptive XML Indexes .......................................................................................................... 25
       3.2.4. Conclusion .............................................................................................................................. 26
   3.3. STRUCTURAL-JOINS XML INDEXES ............................................................................................... 26

4. NODE ENCODING APPROACHES ....................................................................................................... 27
   4.1. PREFIX ENCODING APPROACHES .............................................................................................. 27
   4.2. REGION-BASED ENCODING APPROACHES ............................................................................... 28
   4.3. CONCLUSION ............................................................................................................................... 30

5. PATH ENCODING APPROACHES ..................................................................................................... 31
   5.1. PATH-ENCODING IN XML-TO-RDBMS MAPPING ...................................................................... 31
   5.2. NATIVE XML INDEXES USING PATH SUMMARY ......................................................................... 32
   5.3. CONCLUSION ............................................................................................................................... 33

6. SEQUENCE-BASED INDEXING APPROACHES .................................................................................. 35

7. FEATURE-BASED INDEXING APPROACHES .................................................................................... 37
   7.1. WHAT IS SIMILARITY SEARCH? ..................................................................................................... 37
   7.2. IMPLEMENTATIONS AND DATA-STRUCTURES ........................................................................... 37
   7.3. XML FEATURE-BASED INDEXES ................................................................................................. 39
   7.4. CONCLUSION ............................................................................................................................... 42

8. MOTIVATIONS AND A RESEARCH PROPOSAL ................................................................................ 43
List of Figures

FIGURE 2.1: SAMPLE XML DOCUMENT (SHEFUNI DATABASE) ................................................................. 10
FIGURE 2.2: AN XML ELEMENT DIAGRAM .......................................................................................... 11
FIGURE 2.3: A DTD FOR SHEFUNI DATABASE ...................................................................................... 13
FIGURE 2.4: AN XML SCHEMA FOR SHEFUNI DATABASE .................................................................. 14
FIGURE 2.5: CONVERSION BETWEEN RDBMS, SEMI-STRUCTURED, AND XML DATA REPRESENTATION ... 15
FIGURE 2.6: A TREE GRAPH FOR SHEFUNI DATABASE ......................................................................... 16
FIGURE 2.7: XPATH2.0 AXES ................................................................................................................ 18
FIGURE 4.1: A DEWEY ENCODING (EXAMPLE) ................................................................................... 28
FIGURE 4.2: [CKM’02] ENCODING (EXAMPLE) ................................................................................... 28
FIGURE 4.3: AN EXAMPLE OF A REGION-BASED CODING ALGORITHM ................................................. 29
FIGURE 5.1: RELATIONAL SCHEMA FOR XParent .............................................................................. 32
FIGURE 6.1: AN XML TREE AND ITS STRUCTURED-ENCODED SEQUENCE ........................................... 35
FIGURE 6.2: A QUERY REPRESENTATION USING [WPFY’03] TECHNIQUE ........................................ 35
FIGURE 7.1: EXAMPLE: FEATURE-GRAPH MATRIX .............................................................................. 38
FIGURE 7.2: MULTI-DOCUMENT XML DATABASE AND ITS PATHS & TOKENS SETS ............................... 40
FIGURE 7.3: A BITMAP INDEX FOR THE DATABASE IN FIGURE 7.2 ..................................................... 40
FIGURE 7.4: A BITCUBE INDEX FOR THE DATABASE IN FIGURE 7.2 .................................................... 40
FIGURE 7.5: A BIBLIOGRAPHY XML DATABASE .................................................................................. 41
FIGURE 7.6: A BI-SIMULATION GRAPH OF THE XML TREE IN FIGURE 7.5 .......................................... 41
FIGURE 7.7: MATRIX COMPUTATIONS OF THE BI-SIMULATION GRAPH IN FIGURE 7.6 ......................... 42
FIGURE 8.1: A FRAMEWORK STRUCTURE FOR THE NEW FEATURE-BASED INDEX .......................... 46
FIGURE 8.2: AN XML TREE ................................................................................................................ 47
FIGURE 8.3: PARENT-OF RELATIONSHIP MATRIX OF FIGURE 8.2 .......................................................... 47
FIGURE 8.4: CHILD-OF RELATIONSHIP MATRIX OF FIGURE 8.2 .......................................................... 47
FIGURE 8.5: ANCESTOR-OF RELATIONSHIP MATRIX OF FIGURE 8.2 .................................................... 47
FIGURE 8.6: DESCENDANT-OF RELATIONSHIP MATRIX OF FIGURE 8.2 .............................................. 47
FIGURE 8.7: THE MASTER FEATURE-BASED MATRIX FOR FIGURE 8.2 ................................................. 48
FIGURE 8.8: PARENT-OF RELATIONSHIP MATRIX OF FIGURE 8.2 (AFTER INSERTING I) ....................... 50
FIGURE 8.9: CHILD-OF RELATIONSHIP MATRIX OF FIGURE 8.2 (AFTER INSERTING I) ......................... 50
FIGURE 8.10: ANCESTOR-OF RELATIONSHIP MATRIX OF FIGURE 8.2 (AFTER INSERTING I) .................. 50
FIGURE 8.11: DESCENDANT-OF RELATIONSHIP MATRIX OF FIGURE 8.2 (AFTER INSERTING I) .......... 50
1. Introduction

Since February 1998, the eXtensible Mark-up Language (XML [W3C1] [W3CS1] [ABS’00]) has become the standard medium for data representation and exchange over the web [ABS’00]. As a result, the size and the number of XML's users increased dramatically, and the literature to bring this new technology up to acceptable maturity levels is extremely large including proposals for: XML query languages (e.g. Xpath [W3C4], XQuery [W3C3] and XML-QL [ABS’00]), mapping XML to relational databases (examples are [YDF’04] [BP’05] [SKT’01] [ACLLF’06]), commercial XML-enabled databases (such as OpenXML package in SQL-Server [R’05] and XML data type in IBM-DB2 [NL’05]), native XML databases (e.g. Natix [FHKMNW’02], Timber [JACLNPSSWWY’02], and DB4XML[SVMA’04]), and XML data accessibility and security (e.g. Compressed Accessibility Model “CAM” [YSLJ’04], SQL-like XML security model [G’04]). Among this literature, much has been produced by the database community to build up the XML database technology in terms of query optimization techniques and XML storage managements. This article aims mainly to review recent approaches for indexing XML databases.

The existing XML query optimization techniques can be divided in to two main categories. The first category uses the mature relational database management system to store and manipulate the XML data through what are so-called XML-to-RDBMS mapping approaches [YDF’04] [BP’05] [SKT’01] [ACLLF’06] [FK99] [LC’01] [LY’06]. The main concern of such mapping approaches is to find the best relational representation for the XML which preserve the whole XML semantics while minimizing the cost of the mapping process in terms of processing-time and storage-space. The process of XML-to-RDBMS mapping is associated with the problem of the document’s order semantic which can be answered by using node’s labelling techniques. For this, the existing node's labelling schemes are categorised into two main categories; the prefix [TVBSSZ’02] [CKM’02] [KMS’02] [YLML’05] and the region-based [ACLLF’06] [ZNDLL’01] [DTCO’03] [YASU’01] encoding algorithms. Approaches from each category –in turn- are subcategorized depending on the used labelling methodology, and the strengths and weaknesses of each subcategory is criticized.

The second category of the XML optimization techniques includes native XML indexes [WPFY’03] [KSBG’02] [WPFY’03] [ZOIA’06a]. Similar to XML-to-RDBMS approaches, based on the use of document's structure-encoding methodology; the native XML indexes are grouped into: path-encoding approaches [BFM’05] [YDF’04] [YASU’01] [CMS’02] [KSBG’02] [CLO’03], sequence-based approaches [WPFY’03], and feature-based approaches [YYH’05] [ZOIA’06a]. For each group the advantages and disadvantages are discussed.

The outcome from above mentioned discussion is used to motivate a proposed feature-based XML index in Section 8. This feature-based approach aims to improve the performance of the XML query processing by addressing several limitations found in the existing XML indexes, such as the higher processing-costs in terms of time and storage space, and the index update operations. Such improvements might be achieved by employing some techniques from both the database technology and graphs/matrices theories. Designing, testing and evaluating the proposed feature-based XML index is the subject of ongoing research.

The rest of this report is structured as follows: Section 2 gives a brief definition of XML, outlines the XML document’s components (e.g. elements, attributes, DTD ...etc), and introduces the notion of XML databases and its related technologies (e.g. XML data model and XML query languages). In Section 3, the XML indexing problem is introduced, and various indexing requirements are outlined for three different groups of XML indexes (i.e. indexing document-centric databases, memory-based indexes, and structure-based indexes). Sections 4, 5, 6 and 7 analyse four different groups of structure-based XML indexes. Specifically, Section 4 discusses the node's labelling approaches in RDBMS implementations while the Section 5 discusses the path-encoding approaches in both the relational XML implementations and the native XML implementations. Section 6 and 7 respectively explain the sequence-based indexes and the feature-based indexes. As mentioned above, Section 8 identifies some common problems in the existing indexing proposals and discusses a solution opportunity for them using a feature-based approach. Finally, Section 9 concludes the paper.
2. XML Principles

This section gives a general overview of XML and its related technologies to provide a concrete background for the following sections. The section starts by defining the XML, its origin, and document structure. Section 2.2 discusses some XML schema definition languages while Section 2.3 introduces the inheritance of XML data model from its superior class semi-structured data model. Next, because they are very popular standards for querying XML data and most of the investigated indexing techniques in this review are based on them, two XML query languages (namely XPath2.0 [W3C4] and Query1.0 [W3C3]) are introduced in Section 2.4. The last subsection (Section 2.5) concludes this introductory section and motivates the discussion for the upcoming sections.

2.1. What is XML?

The XML can simply be defined [W3C1] as a general-purpose mark-up language that is used to describe both data and a corresponding structure in machine-human readable format. XML is a simplified-descendant version of the Standard Generalized Mark-up Language (SGML [W3C8]) which was originally designed in 1960 to facilitate information-sharing among heterogeneous systems. XML was initially appeared in 1996 and in February 1998 it was accepted [W3C1] by the World Wide Web Consortium (W3C) as the standard medium for exchanging the data over the Web.

From syntactic point of view, data in XML is tagged with an infinite set of user-defined tags that come in pairs and can be nested to arbitrary levels [ABS’00]. A group of such tagged data, with other components (e.g. comments and processing instruction constructs) compose an XML document; the basic entity for storing XML data in the computers.

To show the basic structure of an XML document, Figure 2.1 presents a sample of XML data containing information about some academic modules taught in the University of Sheffield. The data is contained in XML-elements; that are a series of adjacent and nested encodings which consist of tags (surrounded by angular brackets, e.g. <modules> and <staff>) and values that represent the actual data (e.g. "John" and "Alice"). In addition, an XML document may also include a number of attributes. An attribute is a pair of a name and a value that represents a piece of information which is shared by the element and all its sub-elements and can be used to express new information and/or inter-links different parts of an XML document (or multiple XML documents). For example, the attribute declaration (id = "L100") in line 23 represents and attribute named "id" and its value is "L100". This attribute identifies the corresponding staff element listed between the lines 23 and 26, and is referenced by a special-purpose attribute declared in the line 12. Both XML elements and attributes will be further described in the next two subsections respectively.

In summary, the simplicity of the XML document's structure and the expandability of its vocabulary has given the XML an ability to describe most of the existing data models such as the relational data model...
[OO'01] (see Figure 2.2) and the object-oriented data model [GKP'92]. These facts—about XML—have increased its popularity and attracted researchers to build a concrete XML database technology which already had a wide range of implementations in both the database and the information management communities. The following subsection describes the XML terminology most relevant to the intended research. Readers are directed to [ABS'00] [W3C1] [W3CS1] for a full explanation of the structure of XML documents.

2.1.1. XML Elements

An XML element (see Figure 2.2) is a piece of text that is enclosed within a start tag (identified by `<…>`) and an end tag (identified by `</…>`). It is also the basic component of an XML document which has a name and content. An XML element is named after its start (or end) tag-name which is a user-defined name that must follow some conventional naming rules (e.g. `startTag1` & `endTag1` in Figure 2.2).

From a syntactic point of view, each pair of a start- and an end-tag must be identical. Also XML elements can be repeated and/or nested to any arbitrary depth following a simple syntax; that is, the start- and the end-tag(s) of a contained-element must both exist inside the containing-element (observe the relationship between `startTag1/endTag1` and `startTag2/endTag2` in Figure 2.2). This structure determines the type of any XML element. An XML element can be a simple element (i.e. contains only an atomic value such as string and integers), a complex element (i.e. contains sub-elements) or a mixed element (i.e. contains an atomic value and sub-elements). In Figure 2.1, the `<dept>` element is a simple element, the `<staff>` element is a complex element because it contains two sub-elements while the `<shefUni>` is an example of mixed elements.

In addition to the above syntax, the starting-tag of an XML element can be associated with one or more XML attributes. The use and the syntax of XML attributes are explained in the next subsection.

2.1.2. XML Attributes

An XML attribute is simply a pair of a name and a value that is injected in the starting-tag of an XML element (see Figure 2.2) indicating that all sub-elements (of the corresponding element) share the existence of this attribute and its value. Like element names (i.e. tags), attribute names are entirely user-defined text following the same conventional naming rules of XML elements. However, there are two major issues to be considered when using attributes. An attribute cannot be repeated more than once within a single element definition and its value part must be enclosed in double-quotes as a string regardless its data-type.
In addition, three special-purpose attributes, namely "id", "idref" and "idrefs", are used to support references between different parts (i.e. elements) of an XML document or between different XML documents. Therefore, the benefit of this layout is twofold; a) minimizing the XML document size by preventing data redundancies; and b) reducing the cost of updating such referenced data by ensuring that the update will take place at one position in the XML document. Of course one may notice that such features can be also obtained from the use of user-defined attributes, however, an attribute "cannot be used to express an entity if it has children or if it may appear more than once per parent or if the order matters" [H’05].

As an example, in Figure 2.1, each <staff> element has an attribute called "id" which uniquely identifies the staff member in the XML document. To assign a teacher for a <module> element, its sub-element <lecturer> is associated with an "idref" attribute which points to a <staff> element with an ID equal to the value of the associated "idref" attribute. Unfortunately, the attribute "idref" does not support multiple referencing and thus the "idrefs" attribute is to be used in such situations. For example, if a module is being taught by more than one teacher; then we associate the "idrefs" attribute with the corresponding <lecturer> element, and we list as many teacher IDs as we want in the value string of this "idrefs" attribute.

In theory, XML specifications show that the use of attribute does not enhance the XML data's expressiveness [H’05]. Therefore, most of the XML literature, such as XML-to-RDBMS mapping algorithms (see Section 4 & 5.1) and native XML indexing proposals (see Sections 5.2, 6, & 7), were built on the assumption that every attribute can be transformed as a new sub-element contained in the hosting element.

2.1.3. Comments

Comments can be found anywhere inside XML documents. As in programming languages, XML comments are used for clarifying some parts of an XML document, and they are ignored by most of the APIs (and languages) that manipulate XML documents. Comments have the following syntax:

<!-- this is a comment text -->

In addition, a single comment constructor may span to multiple lines in the XML document. (See line (1) in Figure 2.1).

2.1.4. Other XML Components

Beside elements, attributes and comments, an XML document may contain optional components such as processing instructions (used to pass some instructions to the applications that manipulate XML documents), and schema definition instructions (e.g. DTD and XML Schema). In the next section, the roles of XML schema definitions in the XML database technology are illustrated by describing two standards from schema definition languages (i.e. DTD [W3CS5] and XML Schema [W3C6]). Discussing other XML components are outside the scope of this paper but can be found in [W3CS1, W3C1, ABS’00].

2.2. Schema Definition

In structured databases (e.g. relational databases), the schema definition constrains the layout, definition and typing of the underlying data. The schema must then be defined prior to populating the database and the stored data in the database must obey this schema definition. In contrast, since XML is an instance of semi-structured data [ABS’00], the schema no longer plays this role because semi-structured data is self-descriptive (in terms of layout and definition) and XML data need not be consistent. However, there are many reasons why there may be a need to describe the structure of XML data such as checking for information availability in an XML document, therefore, various schema definition languages have been
proposed and some of them have emerged as essential components of XML language. In this section, in order to complete the basic discussion of the XML technology, the most two popular XML schema definition languages are described; the Document Type Definition language (DTD) and the XML Schema language (XSD) in separate subsections respectively. The following subsection outlines some other schema languages such as XDR, OX and DSD while the last subsection defines the well-formed and the valid XML document requirements.

2.2.1. DTD

The Data Type Definition (DTD) was proposed by W3C as a part of XML language to serve as a grammar and to some extent as a schema for the underlying XML data [ABS’00]. The main purpose of the DTD (see DTD specifications in [W3CS5]) is to describe the structure (i.e. the elements and attributes declarations) of the underlying XML document. The DTD’s declarations are stored either in the XML document it describes, or in a separate file (with “.dtd” extension). The second option is more appropriate in the case where many XML documents need to share the same DTD. In such cases, DTD statements are stored in a global file or URL and documents refer to it by the inclusion the following statement:

```xml
<!DOCTYPE root_element_name SYSTEM "dtd_file_name.dtd">
```

However, a DTD does not offer as many constraints as a relational database schema. For example, DTD lacks the notion of atomic data types as it only allows string declarations (i.e. #PCDATA declaration) [ABS’00]. Also DTD’s do not support range specification such as lookup-lists of values and range domains. Some of these limitations are covered by the XML Schema which is discussed in the following subsection. Figure 2.3 shows a DTD for the XML database found in Figure 2.1.

```
<! DOCTYPE ShefUni [
<!ELEMENT ShefUni (#PCDATA,modules,staffs)>
<!ELEMENT modules (module*)>
<!ELEMENT module (title,credits,sections)>
<!ATTLIST module cid CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT sections (section+)>
<!ELEMENT section (regist,lecturer)>
<!ATTLIST section secid ID #REQUIRED>
<!ELEMENT regist (#PCDATA)>
<!ATTLIST regist secid ID #REQUIRED>
<!ELEMENT lectRef IDREF #REQUIRED>
<!ELEMENT staffs (staff*)>
<!ELEMENT staff (name,dept?)>
<!ATTLIST staff id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT dept (#PCDATA)>
]
```

Figure 2.3: A DTD for ShefUni database

2.2.2. XML Schema

XML Schema [W3C6] [W3CS2] language was adopted as W3C’s Recommendation in May 2001 to avoid several limitations that are faced when using DTD. One of the key factors that strengthen the XML Schema is its support of conventional data typing for XML data [W3C6]. Beside declaring XML elements/attributes, defining the structure and relationships between document components, and supporting data types; the XML Schema supports value-range domains and the occurrence frequency of XML elements in the XML document. Furthermore, XML Schema declarations follow the XML language syntax, support namespaces, and are more human readable than DTD declarations.

An instance of the XML Schema is called an XML Schema Definition (abbreviated as XSD) and, as discussed for DTD’s, can be stored directly in the XML document it constrains or in a separate data file with “.xsd” extension. Figure 2.4 illustrates an XSD for the database in Figure 2.1.
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>

<!--ShefUni (root) element declaration:
<xs:element name="ShefUni"/>
<xs:complexType>
  <xs:sequence mixed="True">
<!--modules element declaration:
<xs:element name="modules">
<xs:complexType>
  <xs:element name="module" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="credits" type="xs:positiveInteger"/>
        <xs:element name="sections">
          <xs:complexType>
            <xs:element name="section" minOccurs="1" maxOccurs="20">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="regist" type="xs:positiveInteger"/>
                  <xs:element name="lecturer">
                    <xs:complexType>
                      <attribute name="secid" type="xs:string" use="required"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                <attribute name="sfid" type="xs:positiveInteger" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:complexType>
        </xs:element>
        <attribute name="cid" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:complexType>

<!--staffs element declaration:
<xs:element name="staffs">
<xs:complexType>
  <xs:element name="staff" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="dept" type="xs:string"/>
      </xs:sequence>
      <attribute name="id" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:schema>

Figure 2. 4: An XML Schema for ShefUni Database

2.2.3. Other XML Schema Languages

In reality, there are many XML schema languages other than the W3C’s DTD [W3CS5] and XML Schema [W3C6] mentioned above. These languages attract less attention from the XML database community because of their limited or specific-domain usages. Regardless of their popularity, most of the existing schema languages have at least as much support for the XML document’s structure as the DTD language. Furthermore, a group of existing schema languages, for example XDR [LTG1] and SOX [W3C9], incorporate support for the basic data types as well as the “string” type which is the only type supported in DTD. However, these examples lack some other aspects of schema definitions such as the explicit-null and the user-defined data-types. These features, plus others such as the uniqueness, key-ness and the inheritance are well-supported in schema languages like DSD [BRICS1] and Schematron [SCM1].

In the literature, a comprehensive analysis of six schema languages; namely DTD [W3CS5], XML Schema [W3CS2], XDR [LTG1], SOX [W3C9], Schematron [SCM1] and DSD [BRICS1] is given by
Lee and Chu in [LC’00]. The authors compare the six schema languages mentioned in terms of syntax, data-typing, components declaration, constraints definition, usability and popularity. Based on this study, the strengths and the weaknesses of any schema language should not be ultimate; rather the judgement should be based on the “philosophy by which each language has been designed” [LC’00]. While some languages might be designed to be more semantic-based, others are meant to be more optimistic. However, as far as the both requirements are concerned, the XML Schema is a superior, and that is why it has been widely accepted in the XML database context [LC’00].

2.2.4. Well-Formed & Valid Documents

An XML document is said to be well-formed if –and only if- it satisfies the following three conditions:

1. Start-tag and end-tag of all elements must be matched (i.e. identical)
2. Sub-elements must be nested properly inside their parents (i.e. sub-elements must be opened and closed within the boundary of the containing elements)
3. Attributes associated with an element must be uniquely defined

Being well-formed is enough for an XML document to be modelled as a node/edge labelled tree (see Section 2.3 for XML data modelling) and therefore can be parsed with existing XML parsers. In addition to the well-formed conditions, an XML document is said to be a valid document if –and only if- there is a schema definition attached to it (e.g. DTD), and the data contained in the document obeys this schema definition [W3CS1] [W3C1] [ABS’00].

Although it weakens the flexibility of semi-structured data, XML document validation plays a major role in several XML implementations such as query optimizations techniques [DTCO’03] [BGK’06] [EDR’05], and XML-to-RDBMS mapping algorithms [YDF’04] [R’05] [ACLLF’06].

![Table](image)

(a) Staffs Database (Relational)  
(b) Staffs Database (semi-structured)

```
{staffs:  
  {staff: {id: L100, name: John, dept: DCS}, 
   staff: {id: L200, name: Alice, dept: MATH}, 
   staff: {id: L300, name: Andrew, dept: DCS}}}
```

(c) Staffs Database (XML)

```
<staffs>  
  <staff id="L100">  
    <name>John</name>  
    <dept>DCS</dept>  
  </staff>  
  <staff id="L200">  
    <name>Alice</name>  
    <dept>MATH</dept>  
  </staff>  
  <staff id="L300">  
    <name>Andrew</name>  
    <dept>DCS</dept>  
  </staff>  
</staffs>
```

Figure 2.5: Conversion between RDBMS, Semi-structured, and XML data representation

2.3. XML Data Model

As mentioned in Section 2.1, XML gained its popularity in the context of database technology because of its self-descriptivism, simplicity, and machine/human readability [W3CS1] [ABS’00]. Because the data is always a valuable source of information, it is essential that it be securely stored, efficiently searched and retrieved, and easily updated. As a result, there has been much research into bringing XML’s performance up that of the current established database technology.
The content of this section aims to give a concrete background to the subsequent discussion of XML data modelling. Therefore, the derivation of the XML data model from its superior semi-structured data model is discussed in Section 2.3.1. Next, Section 2.3.2 discusses the relationship between the two models while Section 2.3.3 describes a graphical representation form for XML data.

2.3.1. XML and Semi-structured Data Model

XML data is said to be an instance of the semi-structured data [ABS’00] where there is no clear cut line between the data and its structure (schema). The definition of the schema is mixed with the data in a serialized text to suit users and applications needs. Therefore, the schema in semi-structured data no longer constrains the data representation as this is the case in the relational data model. Consequently, this gives semi-structured data a flexible format, and hence makes it a suitable model for transferring data between heterogeneous systems [ABS’00].

Another important factor that gives the semi-structured data model strength in today’s database technology is its ability to represent data from other models such as the relational data model. There is a simple and direct mapping to represent an instance of relational data into a corresponding version of semi-structured data (and consequently into an XML version). However, the relational-to-XML conversion process may not be an easy task especially when the efficiency is an issue. An efficient XML design has to satisfy several concerns such as minimal data-storage and optimal XML query supports. Furthermore, the relational-to-XML conversion process results in some information loss such as data-type constraints, internal triggers, data-integrity keys and other constraints; and the cost of such issues must be kept to a minimum. Figure 5(a, b and c) exemplifies a simplest relational-to-XML conversion results.

2.3.2. Order

Although XML and semi-structured data models share several features (e.g. flexibility, self-descriptivism, simplicity, and machine-human readability), there is a major conflict between them in the support of “order” [ABS’00]. While the order is irrelevant in semi-structures representation as the semi-structured data is expressed in terms of set(s), the order is important in an XML context because XML was proposed originally as a document mark-up language. So, ignoring the “order” while processing these documents leads to a semantic loss.

In the XML context, the notion of “order” is applied to the elements but not the attributes. For example, if an element “E” contains two sub-elements, say “E1” and “E2”, and two attributes, say “A1” and “A2”,
then the position of “A1” and “A2” can be interchanged without affecting the document’s semantics while the swapping of “E1” and “E2” positions yields a different meaning.

As a result, managing the "order" produces an extra complexity during the XML database manipulation (i.e. querying and updating XML data). For example there is a need for note-keeping techniques during the XML-to-relational mapping process in order to preserve the structural semantics in XML documents. A detailed investigation on this problem, its effects, and proposed solutions is given in Sections 4.

2.3.3. XML Tree and DOM

Most of the literature represents XML data as an ordered, node-labelled or edge-labelled, rooted tree graph. In a node-labelled representation, the graph consists of nodes and edges. Each node represents an element, attribute or atomic content which is always of type “string”. The graph is initiated from a single node representing the root element. Child elements are then connected with directed edges from the parent node to the child node. The leaf nodes always contain atomic data while the inner nodes hold the tags and attributes names.

This representation varies slightly in the literature depending on the logic of the targeted implementation. For example, edges are labelled with the tag or the attribute names while the nodes are left empty, or incorporated with an invented attribute that reflects the document's order. This is called an edge-labelled graph.

Regardless of the representation used; most XML implementations (e.g. XML query languages) use the following conventions [W3C7] to model the XML data during data processing:

- The top-level node (root) is the 1st node in the document
- The node precedes its children and all its descendants in the document order
- If an element node is associated with namespace nodes, the namespace nodes are ordered after their associative element node.
- The order of the attribute nodes of an element follow the order of the element node and –if found- the associated namespace nodes
- Sibling nodes of an element node come after the element’s children and descendant nodes
- The children property of an element node determines the relative order of the element’s children (the order among the siblings)

A computer-based version of the tree model described in this section is the Document Object Model (DOM [W3C10]). DOM is a platform-independent tree model proposed by W3C to access XML documents in the computerised systems such as W3C’s Java DOM API [SUN1]. In the DOM tree model, an XML node belongs to one of the following categories: element nodes, attribute nodes, and textual-content nodes. The attribute names and the tag names are used to label the attribute and the element nodes respectively while values (i.e. textual contents) are contained in the textual nodes. Each category of XML nodes has its own methods and events that allow an easy access for computer-based implementations to parse an XML document and identify its components for further manipulations. The tree graph in the Figure 2.6 represents the ShefUni database (given in Figure 2.1) using the DOM tree representation. In the following section, the discussion in the rest of this review is motivated by describing some of the well-known XML query languages that use this data model to process XML data.

2.4. XML Query Languages

The most important reason for storing data is to use it again in one or more of the following situations:

- Retrieve the whole or a part of data,
• Re-format the data layout to suit special needs and different applications,
• Construct new data (e.g. produce statistics) from existing data, and
• Update data when circumstances change.

Query languages—in the database context—perform the first three tasks. On some occasions, query languages are extended to do the fourth task. This section aims to introduce two of the widely-used XML query languages, namely XPath (XPath2.0 [W3C4]) and XQuery (XQuery1.0 [W3C3]). The reason for discussing these query languages is that they are widely used XML query languages and lots of today’s optimization proposals are based on them and hence their relevance to this research. However, the discussion is not intended to be comprehensive, and therefore, readers are directed to the W3C’s specifications for more information (XPath [W3C4] and XQuery [W3C3]). The discussion starts by introducing regular path expression in Section 2.4.1 followed by listing the features and the syntax of XPath2.0 and XQuery1.0 in separate subsections.

<table>
<thead>
<tr>
<th>Axis Description</th>
<th>Abbreviated Syntax</th>
<th>Expanded Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Child node</td>
<td>default</td>
<td>child</td>
</tr>
<tr>
<td>Descendant nodes</td>
<td>not available</td>
<td>descendant</td>
</tr>
<tr>
<td>Current and descendant nodes</td>
<td>//</td>
<td>descendant-or-self</td>
</tr>
<tr>
<td>Parent node</td>
<td>.. “two dots”</td>
<td>parent</td>
</tr>
<tr>
<td>Ancestor nodes</td>
<td>not available</td>
<td>ancestor</td>
</tr>
<tr>
<td>Current and ancestor nodes</td>
<td>not available</td>
<td>ancestor-or-self</td>
</tr>
<tr>
<td>Following node</td>
<td>not available</td>
<td>following</td>
</tr>
<tr>
<td>Preceding node</td>
<td>not available</td>
<td>preceding</td>
</tr>
<tr>
<td>Following siblings</td>
<td>not available</td>
<td>following-sibling</td>
</tr>
<tr>
<td>Preceding siblings</td>
<td>not available</td>
<td>preceding-sibling</td>
</tr>
<tr>
<td>Current node</td>
<td>.. “dot”</td>
<td>self</td>
</tr>
<tr>
<td>Attributes</td>
<td>@</td>
<td>attribute</td>
</tr>
</tbody>
</table>

Figure 2.7: XPath2.0 Axes

2.4.1. Path Expression

Given that the best way for describing semi-structured data is representing it as a labelled-tree graph [ABS’00], reaching a specific position (i.e. accessing data) at this tree graph requires navigation from the root node to the desired node. The Path expression [W3C4] is a powerful technique that enables applications (based on semi-structured data) to access arbitrary positions in the tree by walking through labelled-edges. Each edge in the path is called a “step”. Every two steps—in a path—are separated with a dot (i.e. “.”) indicating that the left-hand side step precedes the right-hand side step. Similarly, some operators such as “|”, “?”, “*”, and “+” are borrowed from regular expressions to express choices, existence, one or more repetitions, and none or more repetitions of a certain node in a path expression.

XPath expressions, the XML notation of path expressions, use slashes (i.e. “/”) instead of dots to separate path steps. In addition, a set of axes [W3C7] were introduced by XPath expressions in order to navigate through the XML tree from/to any arbitrary node in the XML tree. XPath axes can be written in abbreviated or expanded forms. Figure 2.7 lists all possible axes in both forms. In general, XPath expressions are forming the base for all XML query languages.

2.4.2. XPath2.0

This section only describes the features and the syntax of the XPath2.0 query language for the purpose of introducing the language. A comprehensive discussion about XPath2.0 technical specifications can be found in [W3C7] [W3CS3].

Features: [W3C7, W3CS3]
• As a query language, XPath refers to XML Path Language
• Models XML data as an edge-labelled tree
• Is used for addressing and selecting certain parts of XML documents or producing summaries
• Can express complex queries by using predicates which are enclosed in square brackets “[…]”. These predicates must be satisfied before the XPath begin matching the following node in the path (e.g. //staff[@sfid='L100']/dept)
• Has a wide range of operators to deal with data such as Boolean operators (e.g. “or”, “and”), arithmetic operators (e.g. “+”, “mod”), and comparison operators (e.g. “>”, “=”)
• Owns a rich library of functions that manipulate node-sets (e.g. position(), count(node-set), strings (e.g. string(obj), concat(str1,str2[, str*]), Boolean values (e.g. not(boolValue), true()), and numbers (e.g. sum(num_nodeSet), average(num_nodeSet))
• XPath2.0, a W3C Recommendation on 27/Jan/2007, is the latest version of XPath which is a subset of XQuery1.0. [W3C3]

Syntax: XPath queries use the notion of regular path expressions along with the above mentioned XPath axes, operators and functions. Some XPath queries –based on the ShefUni database- are given below.

Examples: These examples are based on the ShefUni database in Figure2.1:

Query: Q1: This query returns contents of the lecturer-name nodes:
//ShefUni/staffs/staff/name/text()
Result: Q1:

    John
    Alice

Query: Q2: This query returns “name” nodes for DCS staffs:
/ShefUni/staffs/staff[dept='DCS']/name
Result: Q2:

    <name> John </name>
    <name> Alice </name>

Query: Q3: This query returns the name node of the first staff in the database:
/ShefUni/staffs/staff[1]/name
Result: Q2:

    <name> John </name>

2.4.3. XQuery1.0

Similarly, this section only describes the features and the syntax of the XQuery1.0 query language for the purpose of introducing the language. A comprehensive discussion about XQuery1.0 technical specifications can be found in [W3C3] [W3CS4].

Features: [W3C3, W3CS4]
• Stands for XML Query Language
• Is an XML query language with some programming language features and SQL-like semantics
• Based on XPath data model
• Has all functionalities, libraries and capabilities of XPath2.0 (i.e. XPath 2.0 is a subset of XQuery)
• Is supported by most commercial RDBMS such as IBM, Oracle and Microsoft SQL-Server
• In addition to XPath2.0 capabilities, XQuery supports FLWOR [W3CS4] expressions (FLWOR is an acronym for "For, Let, Where, Order by, Return")
• XQuery1.0, a W3C Recommendation on 23/Jan/2007, is the latest version of XQuery
**Syntax:** There are two different syntaxes for XQuery1.0 query language. The basic syntax is XPath because XPath2.0 is a subset of XQuery1.0. In other words, all XPath2.0 expressions are valid XQuery1.0 expressions. The second XQuery1.0 syntax is the FLWOR expressions which is influenced by some functional programming and SQL-like features. As a result, extra rules are applied to validate the syntax of XQuery1.0 expressions (quoted from [W3CS4]):

- XQuery is case-sensitive
- XQuery elements, attributes, and variables must be valid XML names
- An XQuery string value can be in single or double quotes
- An XQuery variable is defined with a $ followed by a name, e.g. $bookstore
- XQuery comments are delimited by (; and ;), e.g. (; XQuery Comment ;)

**Examples:** All XPath examples (in Section 2.4.2) are valid XQuery1.0 expressions and they produce exactly the same results. Furthermore, the queries Q1 and Q2 (from Section 2.4.2) can be re-written in the form of FLWOR expressions with additional task that changes the order of the results.

**Query:** Q1: This query returns contents of the lecturer-name nodes:
```
for $n in doc("ShefUni.xml")/ShefUni/staffs/staff/name/text()
order by $n
return $n
```
**Result:** Q1:
```
Alice
John
```

**Query:** Q2: The following query creates new elements that list out all staff names from DCS department:
```
for $n in doc("ShefUni.xml")/ShefUni/staffs/staff
  where $n/dept = ‘DCS’
order by $n/name
return $n/name
```
**Result:** Q2:
```
<name> Alice </name>
<name> John </name>
```

**2.5. Summary**

This section (Section 2) consisted of a general overview of XML and some of its related technologies including XML data modelling and XML query languages. As has been explained above, XML is an emerging standard media for transferring data over the Web and exchanging the information between heterogeneous systems. Furthermore, XML is used in many fields and the amount of the information stored in XML format has become incredibly large in a short period of time. Computer and information technology scientists have suggested that XML databases are about to replace the existing conventional databases (e.g. including relational and object-oriented databases) because of its simplicity and flexibility in most fields of information technology. Therefore, the recent database literature aims to bring XML database technology up to the level of maturity of existing conventional databases. This maturity includes reliable XML storage management, efficient XML processing techniques, and trustable XML data warehousing tools. Among these requirements, query optimization techniques have become essential tools for processing large-scale XML databases efficiently. The rest of this review discusses the existing XML indexing techniques for the purpose of investigating further enhancements in such query optimization tools. The discussion ends by proposing a new indexing technique which possibly could improve the performance of a wide range of XML queries.
3. Indexing XML Data

Data is stored in order to be used when needed. The process of storing and retrieving data is associated with techniques such as data storage management and data indexing techniques. Indexing databases has become of critical importance due to the increase in their size. In terms of XML databases, storage management and indexing problems are even worse because of the irregularity of the data and the lack of data-typing [ABS'00]. This review mainly investigates XML indexing techniques in both XML-native platform and XML-enabled platform.

To present the strengths and the weaknesses of XML indexes clearly, the discussion is based on three different aspects. First, XML indexing proposals might be evaluated according to the type of the documents indexed. So, there are indexes designated to index data-centric document, for example Vist [WPFY’03], XSeq [MJCW’04] and FIX [ZOIA’06a]. On the other hand there are indexes designed to index document-centric XML databases such as [XP’05] [YL’06] [BG’06] which implement Information Retrieval (IR) techniques to process XML contents. The framework of some recent document-centric indexes is outlined in Section 3.1. Sections 4 to 7 evaluate data-centric indices.

The second aspect of this discussion of XML indexes is based on the index residency. There are temporary indexes that to be built on-the-fly during the query execution (e.g. [ZNDLL’01] [BKS’02]). This type of index has an excellent response time for structural-joins queries because they avoid the expensive cost of I/O operations as they reside in the main memory. Unfortunately, memory-based indexing techniques lack the scalability for large XML databases, and hence the need for disk-based indexing algorithms (e.g. [WJWLLL’05]). Section 3.2 describes the technology of memory-based indexes with some recent proposals from the literature.

Alternatively, XML indexing algorithms can be evaluated on the basis of how the XML structural relationships are encoded in the index file [ZLC’04]. Indexing techniques of this type have been grouped into four categories. These categories are; node’s encoding techniques, path encoding techniques, sequence-based indexing techniques, and feature-based indexing techniques. Section 3.3 outlines the four categories in general, while Section 4, 5, 6 and 7 respectively discusses each category in detail. An important observation about this class of indexes is that they pay less attention to indexing the contents (i.e. values). This is obvious because the most important issue in querying XML data is to find the proper path to the data despite the irregularity of the XML. The major part of this report is based on a discussion of these indexes because of their relevance to the proposed research in indexing XML data which is outlined later.

3.1. Processing Document-centric XML

In this section I discuss the notion of processing content-oriented XML documents. Unlike the processing of data-centric XML documents, which is intensively investigated in this review because of its relevance to research, only the most significant aspects of querying and indexing document-based XML data are outlined for the purpose of comparing and contrasting them with the data-centric approaches. In the following sections, the need for dedicated content-based XML indexing techniques is discussed followed by several solutions for processing content-based XML documents (Section 3.1.2) and a conclusion in Section 3.1.3.

3.1.1. Querying Content-oriented XML: A Problem Definition

Querying content-oriented XML documents is different from processing data-centric and/or plain-text documents. Data-centric documents can be viewed as a representation of flexible relational tables where the document structure is constrained by the use of elements and attributes [W3C1] [KMRS'04]. Therefore, this type of XML documents is often processed via SQL-like query languages such as XPath
and XQuery. On the other hand, text in document-centric documents is more or less narrative, and loosely structured, with the elements’ order being of more significance. However, it is unlike plain-text documents where there is no structure to be considered during the query process, and therefore, existing information retrieval (IR) techniques are used to process such documents [L’06] [KMRS’04]. Indeed, the mixed nature of content-based XML document has increased the need for a hybrid querying approach where the structural nature is processed by the means of database aspects and the textual nature is managed by employing the IR technology [KMRS’04].

It has been shown that neither structural query tools (for example XPath) nor textual IR approaches can independently process query XML document-centric databases [KMRS’04] [L’06] [FG’01]. In the XPath query language for example, the query processor uses an exact-matching approach to return results. This is obviously not ideal for querying document-centric XML data because firstly, textual content in such document is too big to be embedded (e.g. as predicates) in a query expression. Secondly, textual content contains large amount of redundant information (i.e. non-keywords) which has no obvious and direct role in a determination of the query result, and consequently could lead to false-positive responses. Last but not least, any XPath processor is designed to return the entire matching subtree(s) corresponding to the document’s order. A proper design of an IR system should return the most relevant part of documents with highest relevance ranked first [KMRS’04]. While the “contains” operator in XPath2.0 [W3C4] solves a small portion of the keyword-search problem, there is still a need for other operators to incorporate the document constraints (such as tag and attribute names as well as documents hierarchal structure) during the querying process life-cycle starting from matching phase, through selecting the most-relevant results and ending to results’ ranking phase.

In comparison, IR approaches perform no better than XML query languages for processing document-centric XML data. Guo et al. [GSBS’03] have identified three major challenges to be addressed when using IR keyword searching (KWS) techniques to process XML documents in general and document-centric XML data in particular. First, the result returned by KWS techniques is no longer the entire matching document as it is in traditional KWS. In XML context, a KWS technique has to consider the nested-hierarchal nature of XML documents. That is, if one or more keywords exist in an element, it would be more practical to return that element (or its surroundings) instead of returning the entire XML document. The following section (Section 3.1.2) contains detailed discussion on this issue.

Having all matching elements identified, the second challenge is how to rank these results. We should notice that the ranking computation is now based on the element granularity instead of document granularity [SKR’04] [GSBS’03] and this could lead to undistinguishable ranking values as the keywords sought might be found evenly in the returned elements. So, the traditional KWS techniques are not an ideal solution for performing XML keyword searches. In addition, because of the hierarchal nature of XML elements, the results could be meaningless (e.g. selecting an incomplete portion of text or elements), and therefore we require intelligent algorithms to select the most desirable XML fragments. This involves extra computation and accordingly increases the query’s processing cost. Finally, the proximity computation is a straightforward process for plain-text documents which can be directly derived from the keywords’ distances in the document. In the XML context, however, the proximity computation is more complex. A two-dimensional proximity metric is required to consider both the keyword distance within the entire XML document, and the distance between the keywords’ hosting elements and their ancestors.

In summary, content-oriented XML documents can be processed by mixing techniques from both the standard database-querying technology and the traditional IR technology. Implementing such hybrid approaches requires a consideration of using: a proper query-document matching technique, an intelligent output-selector algorithm, and a reliable ranking scheme. The following section investigates several well-known approaches that have been proposed to satisfy these requirements.

### 3.1.2. Content-based Query Approaches: An Evaluation

Many solutions were proposed to query and index content-based XML documents efficiently using a hybrid technique from both the database community and information retrieval technology. In this section various implementations of such hybrid approaches are described. The discussion includes answers for
the following three questions: How to match the query terms with the document terms? How to select and group the results? How to rank the selected results?

XIRQL [FG’01], is one of the earliest content-based XML query languages and was proposed by Fuhr and Großhoehann in 2001. It is built on the top of XQL [W3C5], a predecessor XML query language that allows flexible conditions on both the XML structure as well as the contents. The main goal of XIRQL is to extend XQL by employing traditional IR keyword search techniques with the structural-semantic support in mind. To do this, XIRQL starts by treating XML leaf nodes as atomic query-able and indexable units, and expands them to what is called index-objects. The idea of forming such index-objects is borrowed from FERMI multimedia model [CMF’96] in which they are used to retrieve the most relevant text from the searched documents to answer a query. In XIRQL, index-objects are a set of distinct subtrees that represents meaningful output for a content-based query. Therefore, the use of these index-objects is twofold: firstly, they are used in the elements’ weighing process and secondly, the matching index-objects are returned in the final result. To access these objects during the query’s different phases, XIRQL uses a simple extension of inverted files and treats index-objects as if they were be stand-alone documents in the traditional IR systems. One drawback of this approach is the use of the pre-set objects which often require updating when the XML structure changes. However, prior specification of these index-objects may prevent undesirable results such as returning full documents.

To avoid the use of pre-set index-objects, XRank [GSBS’03] and XSEarch [CMKS’03] use two automated methods to highlight the element nodes to be processed (i.e. compared against query keywords). In XRank, an XML document is partitioned as follows; each partition (i.e. object) includes all nodes that have at least one occurrence of all searched-keywords; such that they belong to the same ancestor. However, to avoid duplicating descendant elements in the higher partitions, XRank conducts a bottom-up partitioning process excluding the descendant partition from appearing its ancestor [GSBS’03]. XSEarch, on the other hand, uses the idea of interconnection [CMKS’03] to partition XML documents. Two nodes belong to the same partition if there are no distinct nodes holding the same element name in the same set of nodes that share the same lowest common ancestor [CMKS’03] [LC’07]. In addition, XRank does not differentiate between the keyword types (i.e. whether it is a text-word or a label-word) when matching keywords. This is also implemented by XSEarch but the later employs more IR techniques, such as tf-idf, to prevent returning unrelated results. In terms of forming and sizing the output, both XRank and XSEarch return the same or a simple variation (such as meaningful or smallest lowest common ancestor element) of document’s partitions used in the matching phase. However, both techniques rank the results slightly different. XSEarch ranking scheme is mostly IR-oriented. It considers factors such as terms distance and frequency [LC’07] while XRank employs the Google’s PageRank [BP’98] method but with the XML elements granularity instead of documents (HTML documents in the case of [BP’98]) granularity [GSBS’03]. One common problem in both XSEarch and XRank is their ignorance of the type of searched-keyword (i.e. whether a keyword is textual word or node-label word) which may produce semantically irrelevant output or cause the return of the entire document especially in the case of keyword-enriched queries.

The problem of semantic-irrelevancy that is resulted from using both textual and node-labels keywords in XML’s IR approaches is investigated in [KMRS’04] and the “about” operator is proposed to enable XPath query language to efficiently query document-centric databases. The “about” operator has the same syntax as the standard “contains” operator [W3C4] but with two more unique features. Beside its standard syntax, the “about” operator allows the formulation the search requirement in terms of mixing the contents and the structure of XML document. Second, it allows the use of best-match querying of content-based XML documents. So, the XML structural semantic still can be preserved by using strict XPath axes while the contents can be searched by injecting one or more “about” operator into the XPath expression. In terms of managing the query output, although it is not explicitly stated in [KMRS’04], it is obvious that the XPath standard library will control the output produced as well as its order. Having said that, XPath is a subset of the W3C standardized XQuery1.0 language; the “order by” clause can be used to re-arrange the XML fragments returned based on calculated criteria throughout other XPath operators. From the database point of view, the “about” extension seems to be ideal solution for querying content-based XML documents. However, such documents contain information mostly of interest to

---

1 The tf–idf (term frequency–inverse document frequency) is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus.

2 Queries that include many keywords from both; the document’s tags vocabulary and the actual data (i.e. leaf-nodes)
XPath/XQuery ignorant users. Therefore, IR-oriented solutions, with XML’s features, are more practical in this context.

A recent solution for the XPath’s “about” extension problem is proposed in [CPCFD’06]. Chu-Carroll et al. suggested that, incorporating some semantic issues from XML Fragments query language [CMMMS’03] into traditional keyword searching approaches will result in a better query performance for document-centric XML databases. They found that applying conceptualization, restriction and relation operations of XML Fragment query language to traditional keyword searching techniques places more constraints on the query and produces more precise results. So, conceptualization operation is used for query-document matching phase while the ambiguity of the keywords is solved via the restriction operation which also used to specify the search-terms that are interrelated by the relation operation [CPCFD’06]. The good feature of this approach is that it is applicable in most of the existing keyword-based query languages. Therefore, it leverages the characteristics of the underlying IR system such are weighing and ranking operations.

3.1.3. Conclusion

This section outlined several approaches for processing document-centric databases. In particular, each technique was evaluated with respect to three main processes: 1) Conducting query-document matching methodology, 2) Partitioning and selecting the most relevant XML data, and 3) Ranking the results according to their relevance. As is notable from the above discussion, all approaches join techniques from the database community with exiting IR techniques to perform these tasks. Since the main focus of this review is an analysis of the process of indexing data-centric databases, detailed discussion of document-centric databases. The following section (Section 3.2) discusses some techniques used in constructing memory-based indexes for XML databases.

3.2. Techniques for Memory-based XML Indexing

This section outlines several state-of-the-art techniques that are used by some of the existing XML indexes in order to minimize the index size so that such indexes can easily fit in the computer’s memory and therefore obtain better query performance by reducing the expensive I/O disk operations. The purpose of this discussion is to seek a better compression technique for the proposed feature-based index to be discussed in later sections. This section specifically describes the XML structural summaries in Section 3.2.1, the notion of adaptive XML indexes in Section 3.2.2, and the notion of selective XML indexes in Section 3.2.3. The discussion is concluded in Section 3.2.4.

3.2.1. Structural Summaries

This technique is widely used in indexing semi-structured data in general and XML databases in particular. The main goal of a structural summary is to eliminate any redundant structural information of the underlying database without losing structural constrains; that is the structural relationship between XML elements such as parent-child and ancestor-descendant relationships.

In the XML context, a structural summary is a smaller version of an XML tree where all paths from the root node to any leaf node in the actual XML tree are preserved in the summary tree. Therefore, at any level in the XML tree, nodes that can be reached by a specific path from the root node are grouped in a single node (called an extent) in the summary tree. The result is another tree with fewer nodes and –most of the time- deterministic navigation at any level of the tree.

An early XML index using this approach is the Strong DataGuide [GW’97] from the LORE project [MAGQW’97]. In this representation, evaluating XML queries that only involve regular path expressions is significantly faster because no recursive and/or backtracked navigation is required to access similar paths at each branch in the summary tree. However, for those queries with a descendant axis or a wildcard node-test, the query evaluation process requires navigating over the full index tree [H’05], and therefore
the benefit from the index is limited to reducing the output size at each query step. Obviously this is not
the case when the actual XML tree contains nodes’ references. In this case, the size of the index
representation becomes larger than the size of the actual XML tree representation because referenced
nodes are repeated as child nodes under the nodes that reference them.

To solve the index size problem caused by the existence of node's referencing, the I-Index [MS'99] tries
to group duplicated nodes into one extent but with multiple edges pointing to it. The resulting summary is
therefore no longer a tree, and this leads to selecting multiple extents for a single query's step. Therefore,
the results must be refined in additional step [H'05].

Further summary indexes (such as 2-Index [MS'99] [H'05]) followed the design of 1-Index to overcome
several existing problems in the Strong DataGuide [GW’97] such as the index-size problem and the
absolute-nature of the index. Most of these trials were affected by some of these problems such as the
trade-off between the index-size and its performance. As a result, some proposals tried to find alternative
methodologies to reduce the index size. Some solutions use the notion of the selectivity to index certain
portions of the XML database such as the frequent accessed information. This idea is explained in the
next subsection.

3.2.2. Selective XML Indexes

XML selective indexes try to employ indexing techniques from the RDBMS literature. In RDBMS, the
database administrator (DBA) can choose certain fields (i.e. columns of a table) to build an index on.
These fields are selected in such a way as to satisfy certain SQL queries that are frequently triggered by
the database's users. Applying this technique to an XML database, a corresponding index no longer
represents the entire XML database. Therefore, the actual database has to be accesses in order to answer
queries.

Another problem to be considered when constructing a selective XML index is the type and the amount of
updates that may happen in the indexed data. As the document's structure is the main concern of any
XML index, the XML data indexed by a selective index must be chosen from the occasionally-updated
structure. This is because when a certain structure is altered by the means of XML update operations, the
new structure will not necessarily satisfy the same set of frequently triggered XML queries. Therefore,
the DBA has to reconstruct the index criteria according to these structure changes.

A well-known XML index of this type is the T-Index [MS'99] [H'05]. The T-Index is summarised in
[H'05] as follows; "... the main idea of T-Index is to establish a structural summary that only covers path
expressions fulfilling one specific template. The template describes the structure of the path expressions
by node tests and placeholders. If the indexing system supports multiple path expressions with different
structures that cannot be summarised with common template, we need several T-Indexes, one for each
template."

Another problem emerges from the above description of the T-Index is reflected in the cost of choosing
an appropriate T-Index to satisfy a certain XML query which triggers the query-rewriting problem
[H'05]. A possible solution for such problems might be achieved by using an adaptive indexing technique
which is able to change its structure incrementally according to the changes in the query workload. The
notion of adaptive XML indexes is described in the following subsection.

3.2.3. Adaptive XML Indexes

As was introduced in the above section, this type of XML indexes tries to update their indexing criteria
incrementally based on the most frequent XML queries. To satisfy this requirement, the database system

---

3 by this I mean the index can be only used to satisfy the absolute regular path queries such as queries that start with the descendant-axis (i.e. // axis)
4 How to rewrite a query in such a way it evaluates to the same set of results but with a fewer processing cost (i.e. query optimization)
(or the DBA) must keep track of all triggered queries along with the changes in the document's structure. There are three obvious drawbacks to this implementation. First, the index will involve expensive computational complexity required to alter the index structure during the query-evaluation process which may result in slowing the query-evaluation process itself. Second, more storage space is required to keep the queries and/or the changes in the document's structure so that the index structure can be updated. Third, the index becomes a sort-of selective index; that is it is concerned with frequent triggered queries and ignores other quires which might involve heavy comparison operations. Two typical examples of adaptive indexes are the Adaptive Path Index (APEX [CMS'02]) and the index proposed in [CLO'03]. These indexes are further discussed in Section 5.2.

3.2.4. Conclusion

This section outlines three techniques that can be used to limit the size of an XML index. These techniques concern reducing the storage space required to store the document's structure rather than the XML data (i.e. value) so that the index can reside in the computer's memory for fast access. These compression techniques divide between complete and partial XML indexes. XML queries benefit less from the partial XML indexes as these indexes are directed to specific classes of XML queries and/or ignore certain portions of XML databases which may be triggered by different interests. In terms of the index proposed later, structural summaries offer a good opportunity -with the conjunction of the XML data representation used by the index- to reduce the index size. Regardless of the compression technique used, the following section introduces a categorization of structural XML indexes based on the methodology used to encode the elements relationships found in the XML documents.

3.3. Structural-joins XML Indices

Structured XML queries (or containment queries) evaluate path expressions (query patterns) to return all matching nodes and/or check the query pattern existence in the XML tree [WJWL'05]. Therefore, an index of this type need to encode as many document’s structures as possible in order to satisfy a widest range of such queries. On the other hand, separate indexes can be created for a single XML document, each of which is used with a specific class of structural queries. The basic framework of any structural index is to encode some structural relationships between XML nodes (elements, attributes, etc) so that the query processor is able to predict results by simply approaching a corresponding index without accessing the actual data file. It has been identified by researchers [ZNDLL'01] [KJKPSW'02] [MW'99] [STZHDN'99] that the parent-child and the ancestor-descendant relationships are sufficient to answer most classes of the structured queries. However the efficiency of the evaluation process varies from one class to another and therefore, as more document’s structures are encoded in the index, better processing efficiency can be obtained [ZOIA’06a].

Based on the algorithm used for encoding the XML’s hierarchal-structure, the structure-based indexes are grouped into four categories. These categories are; simple node-encoding approaches (e.g. [TVBSSZ’02] [SLFW’05] [ZNDLL’01]), path-encoding approaches (e.g. [CLO’03] [CMS’02]), sequence-based approaches (e.g. [RM’06] [WPFY’03] [WJWLLL’05]) and feature-based approaches (e.g. [ZOIA’06a] [YKKC’02]). The following four sections discuss these categories.
4. Node Encoding Approaches

Recall that data in XML documents is modelled as an ordered, node/edge-labelled, and un-ranked tree [ABS’00], thus, during processing operations (e.g. query and update), the XML document’s structure must be efficiently maintained. Keeping the structural relationships up to date within the index file is an important issue for querying XML data efficiently. In reality, most of the existing indexing techniques encode these relationships to speed up the evaluation process of the containment queries [CAO’06]. During the last decade, literature is rich with proposals for encoding the XML’s structural relationships efficiently via what is so-called node-labelling. The main two goals of the existing node-labelling solutions are: a) to assign a unique code for each node in the XML tree, and b) to preserve the nest-hierarchal structure of XML documents (e.g. ancestor-descendant and parent-child relationships) during XML updates [TVBSSZ’02] [SLFW’05] [SKT’01] [CKM’02] [FLSW’03] [EDR’05] [BP’05] [YLML’05]. However, minimizing the re-numbering cost (including the processing time and I/O accessibility) in the case of data updates, and reducing the required storage space to store generated code; are essential requirements for any node-labelling proposal [TVBSSZ’02] [FLSW’03] [KMS’02].

In general, any node-labelling algorithm is either a prefix-based approach or a region-based approach. In this section, some proposals that best describe the characteristics of each category are discussed. In particular, section 4.1 deals with prefix-coding approaches while the following section discusses region-based approaches. Section 4.3 concludes the discussion of node-encoding approaches.

4.1. Prefix Encoding Approaches

Numbering schemes of this type basically generate code consisting of two parts: the prefix part which encodes the preceding node code and the actual-code part which encodes the order of the node among all nodes in the XML tree (or its siblings) using a specific tree-traversal algorithm such as pre-order and post-order traversal algorithms. In this encoding, parent-child and ancestor-descendant relationships are included in the prefix part, while both parts of the generated code form a unique identification for each node.

There are many proposals of this type. A simple but most famous example is the Dewey [TVBSSZ’02] coding algorithm where each node (except the root node) is given a Dewey code that consists of two parts: an incremental local number that reflects the position of the node among its siblings, preceded by the Dewey code of the parent’s node. The two parts of the code are separated by a “.” (i.e. dot). The root-node code consists of one part only because it has no parent. Figure 4.1 shows an XML tree labelled using the Dewey algorithm. Another prefix algorithm is presented in [CKM’02] where the usage of “dot” separators is avoided to reduce the storage space. The algorithm in [CKM’02] works in similar manner to [TVBSSZ’02]: starting from the root node which has a “0” code, its first child is numbered with “0”, then “1” is added to the left-hand side of the first child’s code to form the second child’s code and so on. Of course each child code will be preceded by its parent code as a prefix. Figure 4.2 exemplifies the node-labels that are produced by the coding algorithm proposed in [CKM’02]. It is clear that, the full path - from the root node to any node- can be directly obtained from the node’s label itself in the case of the Dewey algorithms whilst this is not possible with the encoding algorithm presented in [CKM’02]. A better prefix coding algorithm called “Prefix Perfect Binary Tree” (P-PBiTree [YLML’05]) is built on the top of PBiTree [WJLY’03]. The P-PBiTree algorithm preserves codes (of size m-bits) for siblings and descendants of a node when the update occurs in that position. These codes are evenly distributing among the children nodes. Unlike the previous prefix algorithms, P-PBiTree also works when the updates occur in the middle of an XML tree.

An intelligent prefix-based coding algorithm was proposed by Kalpan et al in [KMS’02]. The algorithm uses the idea of a compressed XML tree to reduce the storage space required to store the codes. The algorithm simply identifies all distinct paths from the root node to all possible leaf nodes and builds a virtual-compressed XML tree. Then, a normal prefix-based algorithm is applied to encode the virtual XML tree. Although this technique minimizes the time that is required to access a certain node by
eliminating un-related PATH nodes in the first round, the algorithm complicates the comparison test that is required to identify the ancestor/descendant relationships.

Prefix-based numbering algorithms have several features which make them suitable for some implementations but also there are cases (e.g. depth-oriented XML documents [MO’06]) where their performance drops and therefore better algorithms are required. Like other types of numbering techniques, prefix-based algorithms uniquely identify nodes in the XML tree [TVBSSZ’02] [CKM’02] and preserve the most important structural relationships (namely the parent-child and the ancestor-descendant relationships) between these nodes [YLML’05] [WJLY’03]. In addition, the generated code reflects the direct path (i.e. the linking node-set) between any two descendant-nodes in the XML tree [TVBSSZ’02]. However, in the case of XML updates, the process of re-coding becomes costly although it can be minimized by leaving gaps [TVBSSZ’02] between any successive nodes’ codes. The reason is that the quantity and the place of such updates are unpredictable [W3C1] [YLML’05], and therefore an intelligent updates-prediction algorithm is required to estimate the size of these gaps which may ultimately become exhausted. Furthermore, XML updates are most-likely happen at the lower-levels of the XML tree [THHW’01]. So, the number of re-calculated codes becomes high because prefix-based numbering schemes propagate from top-to-bottom, left-to-right in the XML tree [YLML’05] [HHMW’07] and therefore all descendants and the following siblings need to be re-numbered. Another deficiency of prefix-based algorithms is the amount of storage space required for the generated codes. As the XML tree goes deeper, the length of the generated code increases cumulatively; thus more storage space is required [MO’06].

In summary, the prefix-encoding algorithms perform well in the static XML operations [TVBSSZ’02]. However, there is a need to reduce the amount and complexity of the re-numbering process during the update operations. Also the storage space, which increases in proportion to the depth and the breadth of the XML tree, should be kept small especially when the implementation takes place in memory. Some of these problems are addressed by region-based encoding proposals which are discussed in the following section.

4.2. Region-based Encoding Approaches

The basic idea of any region-based encoding algorithm is to encode the ancestor-descendant relationships (and/or the parent-child relationships) in an XML document by attaching two variables to each node; namely the startID and endID. The first variable stores the node ID of first descendant element/attribute node (which is in most cases the node ID of the node itself). The second variable (i.e. endID) holds the node ID of the most descendant node from the current node [ZNDLL’01]. Node IDs are usually obtained by applying the pre-order traversal [D’82] [LM’01] technique on the document’s tree [ACLLF’06] [ZNDLL’01]. Some proposals add two extra variables to these codes; one of them stores the level of the current node (variable name “level”), and the other stores the document ID (variable name is “docID”) [e.g. ZNDLL’01]. The “docID” variable is required to identify the XML document -to which the current node belongs to- in case of an XML database consists of multiple XML documents. Similarly, the “level” variable works with the conjunction of startID and endID to identify the parent-child relationship. So, the node is eventually associated with the following vector which holds the minimum information to process containment queries [ZNDLL’01]: (docID, startID, endID, level). Some implementations add extra information for different reasons such as pathID which refers the path from
the root node to the current node [e.g. LN’04]. Figure 4.3 illustrates typical region-based codes for an XML document.

![Diagram of region-based coding algorithm](image)

**Figure 4.3: An Example of a Region-based Coding Algorithm**

There are many node-encoding proposals using region-based numbering scheme in the literature such as [ACLLF’06] [ZNDLL’01] [DTCO’03] [YASU’01]. The majority of region-based proposals are used in XML-to-RDBMS mapping implementations to support the evaluation of containment queries over the resulting relational schema. All existing region-based algorithms share the idea mentioned above but they vary in the way in which the vector’s variables are computed. For example, the algorithm in [ACLLF’06] uses the pre-order traversal technique (as described above) to assign the nodes their object-identifiers (i.e. oid). These oid’s are then substituted in the node’s vector to encode the document’s relationships. In contrast, the technique proposed in [ZNDLL’01] builds two inverted-indices for the document’s tokens (Note: a token can be a tag-name or a single textual-content unit (e.g. words, numbers, …etc) with in XML elements/attributes), and uses the token’s location (in the corresponding inverted-index) to code the values of startID and endID. The node’s code in [ZNDLL’01] differs slightly from that in [ACLLF’06].

In [ZNDLL’01], the nodes of the elements and the attributes are associated with (docID,startID:endID,level), and the nodes holding the textual-contents are tagged with (docID,positionID,level); where positionID is the distance (in the number of words) of the token itself from the first token in the XML document. The algorithm in [DTCO’03] is identical to the one in [ACLLF’06] except that it uses the offsets in a string representation of the XML document.

Like prefix-based numbering algorithms, region-based encoding algorithms have features that make them perform better than other algorithms in some implementations, and less efficiently in others. Any region-based algorithm performs better than prefix-based (e.g. [TVBSSZ’02] [YLML’05]) algorithms in terms of the amount of storage space required to store the codes. It is clear from the example in Figure 4.3 that the amount of space for each node’s label depends on the amount of the space reserved for the node’s identifier vectors. If each vector requires m bytes, and the XML tree is of size n nodes; then the total storage space required to store the entire XML codes is m×n. In addition, the identification of ancestor-descendant (and/or parent-child) relationships is much easier in the region-based numbering schemes [ZNDLL’01]. By looking at any two codes; say node1: (doc1,s1,e1,l1), and node2: (doc1,s2,e2,l2); node1 is an ancestor of node2 if (s1 < s2) and (e1 >= e2). Furthermore, node1 is identified to be the parent of node2; if the following condition is additionally applied: (l1 = l2 - 1).

In terms of XML updates, region-based algorithms use the mechanism used by the prefix-based techniques. During the pre-order traversal of the XML tree, some gaps are left between the codes of successive nodes in order to reserve room for any suspected insertions [DTCO’03]. Alternatively, floating numbering might be used, so extra decimal places can be added to the preceding code when coding the newly inserted nodes. However, as in prefix-based techniques, the number and the location of the insertions are un-predictable [W3C1] [YLML’05]; thus, some gaps are eventually consumed and consequently, the re-coding process will be required. Unfortunately, the re-coding process in region-based numbering schemes is more expensive than that in prefix-based numbering schemes. In the former, the coding process propagates from bottom of the XML towards the root node. In practice, because most of the updates (e.g. insertions) happen in the lower levels, the re-coding process affects all the ancestors [LLHC’04]. A similar complexity is encountered when one wants to find the path (i.e. the linking node-set) that links any two nodes in the XML tree. In this situation, it is necessary to walk through the whole
tree of descendant (or ancestor) nodes, checking the parent-child relationships for all successive pairs until reaching the destination node [TVBSSZ’02] [LLCC’05].

To conclude, like prefix-coding approaches, region-based algorithms perform well in the static XML operations although they involve more comparison operations during the process of identifying structural relationships. However, the cost—in terms of storage space—of the region-based numbering approaches is much lower than that of the prefix-numbering approaches. Therefore, reducing comparisons cost and adding the ability to infer paths from the nodes’ code are valid enhancements in the region-based numbering approaches. In [LLCC’05], Lu et al approached such enhancements in their Extended Dewey algorithm that combines the Dewey [TVBSSZ’02] technique with a region-based numbering technique such as [BKS’02] [ZNDLL’01] [KJKPSW’02].

4.3. Conclusion

Zou et al [ZLC’04] summarized the architecture of the all numbering techniques of XML nodes as follows; they “…create indexes on each node by its positional information within an XML data tree. Such index schemes can determine the hierarchal relationships between a pair of nodes in constant time.” They also “use a node as a basic query unit, which provides great query flexibility.” However, these indexes are intensively updated when the underlying database is modified. In terms of query evaluation costs, queries such as involving recursive structural joins, node-encoding algorithms consume more time and storage-space to store the intermediate results. In addition, these techniques pay no attention to indexing XML textual content (i.e. values) except some statistical figures about the appearance of the distinct words in the indexed XML document [ZLC’04]. The next section discusses the architecture of path encoding approaches, which is another class of indexes used to encode XML structural joins.
5. Path Encoding Approaches

Indexing techniques of this type share the idea of creating a path summary for XML data to speed up the process of query evaluation. That is, for an XML document, its path summary includes entries for all possible distinct paths from the root node to any arbitrary node in the document [CLO’03]. An XML query is then evaluated by simply accessing the path summary without the need to process the underlying XML document [BCM’05]. However, as far as a query complexity is concerned, XML path-queries are divided into simple-path queries and complex-path queries (i.e. queries with multiple path-branches). Barta et al [BCM’05] stated that; the path-indexing techniques are ideal for evaluating single-path queries rather than multiple-branches queries (or twig queries). The reason is that, evaluating such complex-path queries involves expensive join-operations that are required to unite results returned by the multiple branches in order to construct the final output that matches the entire query expression. In this discussion, path-summary proposals are divided into two categories: proposals used in XML-to-RDBMS techniques, and proposals used in indexing native XML data. The reason for this separation is that, although both categories use the idea of building path summaries, the first category uses path summaries to design efficient-storage implementations in RDBMS and therefore use RDBMS engines to query XML data, whilst the second category uses path summaries to construct efficient indexing to speed up XML queries in a native XML query languages. In this discussion, Section 5.1 explores some path-summary techniques used in XML-to-RDBMS mapping implementations, and the following section (Section 5.2) discusses some native implementations of path-indexes. The last subsection concludes the discussion on path-encoding approaches.

5.1. Path-encoding in XML-to-RDBMS Mapping

Many techniques have been proposed to store XML data in RDBMS in order to leverage advantages of these relational database engines in storing, querying, indexing and updating data. Recent yet popular examples of these techniques include in-lining [BFM’05], ShreX [YDF’04], and Edge++ [BFM’05]. Among tens of such XML mapping techniques, there are algorithms store XML paths in pre-defined relations in order to preserve the XML structure such as parent/child and ancestor/descendant relationships [YASU’01] [JLWY’02] [LN’04]. While this process has an excellent performance in answering path queries over the stored XML data in RDBMS [KR’01] [TDCZ’02], other issues, such as expensive recursive joins and data updates costs, need to be addressed.

XRel [YASU’01] and XParent [JLWY’02] are two XML-to-RDBMS mapping techniques that use a predefined relational schema of four tables to store both the data and the structure of XML data. However, the two techniques have different layouts of these four tables, especially the number and the design of tables (among the four) that store XML structural relationships. In XRel [YASU’01], only one table, called Path relation, is used to store possible distinct XML paths. The Path table has two fields: the path_ID that holds a short-unique identifier for each XML path, and the Path_Desc which stores all possible unique paths from the root node to any leaf node. On the other hand, XParent [JLWY’02] uses two tables (namely LabelPath and DataPath-also see Figure 5.1) for the same purpose but without storing region information (see Region-based Encoding Approaches in Section 4.2) [YASU’01]. Thus, for a simple path query that only contains parent/child relationships, the evaluation process is straightforward, and no recursive joints are required in order to identify containment relationships. However, when a query contains one or more ancestor/descendant relationships; the evaluation process becomes more complicated, and many joints will be involved on the Path (or LabelPath and DataPath) table(s). In XParent implementation, the problem is worse because the evaluation process requires joining the DataPath relation many times to itself. In addition, during XML update operations, especially operations that affect the document structure; records are subject to frequent updating. However, this may not be a critical problem since most of the updates occur at the leaf nodes (data nodes) rather than upper level nodes. Finally, such mapping techniques are used to minimize the storage space required for storing XML data in RDBMS [KR’01] [TDCZ’02].
INode [LN’04] is another XML-to-RDBMS mapping implementation that stores XML path summaries in a predefined relational schema. The relational schema used in INode has one table less than XParent and XRel (i.e. its schema contains the following tables: Path, Element, and Attribute). The experimental results in [LN’04] have proven that this schema further reduces the number of external joints (and possibly storage space) when performing queries that involve the parent-child relationship or the ancestor-descendant relationship. However, evaluating ancestor/descendant relationships still require expensive recursive SQL queries in the RDBMS implementations.

Unlike XRel, XParent and INode; Monet [SKWW’00] creates a separate relation for each unique path in an XML document to avoid self-joints or recursive queries. However, it is very obvious that there will be many external joints between different path tables even for a simple query. So, this technique performs no better than the above techniques in terms of query evaluation complexity [KR’01] [TDCZ’02].

Figure 5.1: Relational Schema for XParent

To sum up the above discussion, some XML-to-RDBMS mapping techniques create one or more relations for storing all possible distinct paths in an XML documents in order to maintain document’s structure including parent/child and ancestor-descendant relationships. Although this technology reduces the amount of storage space that is required for storing XML data in a corresponding relational schema, and in the same time- preserves its structural relationships; most of the proposed techniques suffer from the internal joins problem and the satisfaction of recursive-queries.

5.2. Native XML Indexes using Path Summary

The underlying technology of XML path-indexing, especially in native XML implementations- is XML path summaries. Structural summaries, another term describing the same technology, have been “proposed to prune the searching space while evaluation path expressions” [CLO’03]. A path summary for an XML document is simply a list of all distinct paths that basically build-up the actual XML document. A collection of all path summaries for an XML document or a set of XML documents (i.e. XML database) are linked together in a tree-like graph called summary graph [HHK’95]. Thus, a summary graph -for an XML document- preserves its structure with fewer nodes and edges. As a result, to evaluate an XML query over an XML database, it would be sufficient to search the corresponding summary graph. Although they share the idea of XML path pruning in order to minimize the search space, different summary-based indexes use different strategies to evaluate XML queries.

Any path summary (i.e. path index) can be either an exact path summary or an approximate path summary [BCM’05]. An exact path summary represents every distinct path of the underlying data tree in the resulting graph summary, whilst an approximate path summary includes only paths to a certain depth. In both cases, the summary is used as a back-end technology for evaluating regular path expressions (most of XPath queries are regular path expressions). In addition, an exact summary can work as a schema for the underlying database which creates an opportunity for a better query optimization [BCM’05]. This class of XML indexes is best represented by the following proposals; 1-Index [MS’99], A(k)-Index [KSBG’02], APEX [CMS’02], and D(k)-Index [CLO’03]. Criticizing these techniques is the aim of this section.
All existing XML path summaries proposals are based on a bi-similar representation of XML data. In such representations, any two nodes are said to be bi-similar if all incoming paths (i.e. paths descendant from the root node) to any of them are identical [HHK’95] [PT’87]. The set of bi-similar nodes are grouped in an equivalence class, and this class forms an I-Node (i.e. Index-Node, some proposals called this node an “extent” node) in the resulting summary graph. Similarly, an edge that connects a pair of I-Node in a summary graph is called I-Edge (i.e. Index-Edge). The 1-Index summary [MS’99] is a complete summary which represents the whole XML tree in the summary graph so that a query can be evaluated without consulting the actual XML tree. However, the notion completeness—in XML path summaries—sometimes results in a large summary graph which makes the searching space almost equivalent to the actual XML tree, and therefore the processing speed is hardly improved [BCM’05].

The A(k)-Index [KSBG’02] was proposed to overcome the large-scaled index problem encountered in the 1-index. It reduces the search space by considering the incoming paths of k-length only. This results in a lower storage space and faster processing but unfortunately the index becomes approximate, and consequently further operations will be required in order to avoid false indications returned by the primary index. An obvious observation about A(k)-Index is that there is a trade-off between the size of the index, which depends on the k-factor, and its performance. Bigger values of the k factor minimize additional processing; but on the other hand increase the index size and consequently increase the processing time.

So far, two main problems have been identified in both 1-Index and its derivation A(k)-Index. These problems are: the big index-size and its inefficiency; and they are both caused by the fact that the underlying summary graphs are static, that is, indexing criteria are not adjustable according to the query workload and/or the indexed data-size. Because of this, a workload-aware path index, called APEX [CMS’02] was proposed by Wan et al in 2002. The APEX index enhances the summary structure by: 1) incorporating a hash index for efficiently evaluating the frequent queries, and 2) calling an algorithm which is able to adjust the index according to the query workload. In [CLO’03], Chen et al. further improved the A(k)-Index by adding a technique similar to the one found in APEX which considers changes both in the query workload as well as the source data. So, the D(k)-Index serves as a dynamic summary index that results in a smaller index size and a better performance. In addition, the dynamism of D(k)-Index can be tuned to simultaneously update the index and evaluate queries.

In conclusion, path summaries; that are built on top of bi-simulation graphs [HHK’95] [PT’87], are powerful techniques for indexing native XML databases because they efficiently preserve the document’s structure with low storage space. Once an exact-optimum summary is obtained, a corresponding index becomes an excellent memory-based solution for indexing and querying XML. However, indexes of this class are basically structure-based techniques with a little or no attention to contents (i.e. values) indexing. Therefore, a great effort is required to make them ideal solutions for a wider XML query types that involve value comparisons.

5.3. Conclusion

The above two sections (Section 5.1 & Section 5.2) discussed some typical XML path-indexing proposals. Path indexes have been used in both XML-to-RDBMS mapping and native XML indexing. In the first implementation (e.g. in-lining [BFM’05], ShreX [YDF’04], Edge++ [BFM’05]), all possible paths; from the root node to the leaf-nodes, are stored in pre-defined relational schemas. These relations are then used to 1) speed up XML queries over a relational database engine, 2) reduce the storage space, and 3) avoid expensive structural joins that are required to evaluate branching path queries. Unfortunately, this technology produces several drawbacks such as recursive self-joins (i.e. recursive SQL queries) even for simple path expressions. In addition, updating the underlying XML data dramatically increases the updating processes for the mapped data.

Alternatively, path summaries have been used to construct native XML indexes (e.g. 1-Index [MS’99], A(k)-Index [KSBG’02], APEX [CMS’02], and D(k)-Index [CLO’03]). All existing path summaries are using the notion of bi-simulation graphs [HHK’95] [PT’87]. This class of XML indexes demonstrated a noticeable improvement in performance in terms of speed and efficiency. However, there is a trade-off
between the size of a path summary index and its performance. The problem could be solved by designing a flexible path summary that can adjust its size according to the query workload and the database size. Additionally, because of its efficiency in describing XML documents, and its noticeably smaller size with respect to actual XML trees; bi-simulation graphs can be employed in designing efficient XML feature-based indexes. Features-based indexes framework is the subject of Section 7. The following section (i.e. Section 6) explores sequence-based indexing approaches.
6. Sequence-based Indexing Approaches

Literature shows that an XPath query can be modelled as a tree structure similar to XML trees [WPFY’03]. It has also been shown that if an XML query tree is a subset of an XML document tree, then the query will return some results when it is processed against the document. This technique, therefore, can be used to predict the query results’ existence in the XML document. One way to implement such technique is to convert both the XML query and XML document into sequences and use the well-established sequences matching techniques to obtain query answers (e.g. [RM’06] [PK’05] [MJCW’04] [WPFY’03]). The good feature of sequence-based indexing techniques is that they use the entire query tree as one unit, thus the expensive join operations are avoided during the query evaluation while the structural relationships are preserved. This section discusses the performance of this class of XML index using some examples from the literature.

<table>
<thead>
<tr>
<th>Label</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>staffs</td>
<td>S</td>
</tr>
<tr>
<td>staff</td>
<td>F</td>
</tr>
<tr>
<td>name</td>
<td>N</td>
</tr>
<tr>
<td>dept</td>
<td>D</td>
</tr>
<tr>
<td>&quot;John&quot;</td>
<td>v1</td>
</tr>
<tr>
<td>&quot;DCS&quot;</td>
<td>v2</td>
</tr>
<tr>
<td>&quot;Alice&quot;</td>
<td>v3</td>
</tr>
<tr>
<td>&quot;MATH&quot;</td>
<td>v4</td>
</tr>
</tbody>
</table>

\[ T = (S,\varepsilon), (F, S), (N, SF), (D, SF), (v2, SFD), (F, S), (N, SF), (v3, SFD), (D, SF), (v4, SFD) \]

Figure 6.1: An XML Tree and its Structured-encoded Sequence

To illustrate how this class of indexing techniques works, consider the technique presented in [WPFY’03] for example. The XML tree in Figure 6.1 is firstly converted into a sequence that is shown at the bottom of the same figure. On the other hand, an XML query listing staffs from the "DCS" department is expressed into four different formats (i.e. verbal, XPath, tree-form, and sequence-form) in Figure 6.2. To evaluate this query, the query processor will search for the query’s sequence into the document sequence. If the query sequence is non-contiguous sub-sequence in the document sequence, then results are returned by the query, otherwise the query evaluates to an empty set. In this example, the query sequence is a non-contiguous sub-sequence (see the underlined entries in Figure 6.1) in the document sequence. Therefore, some results will be returned by the query processor.

**Verbal Form:**
Find all staffs from "DCS" department

**XPath Form:**
/staffs[dept = 'DCS']

**Sequence Form:**
\[ Q = (S,\varepsilon), (F, S), (D, SF), (v2, SFD) \]

**Tree Form**

Figure 6.2: A Query Representation using [WPFY’03] Technique
A superior feature for this approach over the other classes of indexing techniques, especially Node-encoding approaches (e.g. [ZLC’04] [KMS’02]), is that it incorporates a deeper consideration of indexing textual contents (i.e. values). To do so, the indexing algorithm injects invented textual-nodes into XML/query sequences after calculating their labels using a simple hashing [S’07] index. Also the required storage space to store the generated document’s sequence is linear in relation to the XML tree size (i.e. number of nodes), which is less than the storage space required to store codes generated by Node-encoding index algorithms [BG’06].

Although they speed the query pre-evaluation and eliminate the expanse of structural joins, this class of XML indexes have two major limitations. Firstly, the XML sequence –that is generated by any sequence-based algorithm- will be reconstructed frequently because of: 1) the updates in the underlying XML tree structure (e.g. inserting new nodes), and 2) the updates in the textual-contents which result in their coding updates. While the first problem causes a major change in the generated XML sequence, the second problem can be avoided by using a simple mapping list that matches every textual entry with a fixed code. The second limitation in the sequence-based indexes is the use of hashing algorithms to encode the textual contents (i.e. generate codes for the textual-nodes in the XML document). In this, the hashing algorithm simply calculates a hash key for every single textual node, and uses these keys in the XML sequence. Hence there is always infinite set of textual nodes in each document; the consequence is that, the hash list grows very quickly, and eventually the hash fetching process itself becomes slow.
7. Feature-based Indexing Approaches

Indexing and querying algorithms of this type often use similarity search techniques [WDJLCL’07] [WSB’98] to compare a set of features of a query against a similar set of features in the database in order to evaluate queries. In the XML context, feature-based indexing algorithms encode one or more structural features (i.e. structural relationships such as parent-child relationship) of the XML query pattern on one side, and the same set of feature of the XML documents (i.e. XML database) on the other side, and then use this information to check the query’s existence in the XML database [YKKC’02] [ZOIA’06a]. The actual result is then obtained by conducting an additional traversal process on the actual XML data. Depending on the class of targeted XML queries, the type and the amount of the encoded XML features vary from one technique to another. On the other hand, various data-structures were proposed to represent these features in the main memory or disks such as adjacency-matrix similarity testing [ZOIA’06a], hash-based similarity search [S’07], and ranked-based similarity search [WDJLCL’07].

This section initially defines the term Similarity Search as described in the literature (Section 7.1). Next, Section 7.2 discusses some similarity-searching tools and their data-structure backend along with some implementations used by the information retrieval community. The following sub-section (Section 7.3) explores the literature for some recent feature-based indexing proposals.

7.1. What is Similarity Search?

Wang et al. [WDJLCL’07] refer the term Similarity Search as “… searching a collection of objects to find objects similar to a given query object.” In human perception, two objects are said to be similar if they share specific number of features. The similarity might be an exact- or an approximate-similarity. In the exact-similarity search, the duty of a query processor is to find all identical objects that match the query-object. The main problem of the exact-similarity search techniques is their exponential cost in terms of search time and search space [DL’76]. Therefore, researchers have studied the problem of finding the $d$-distance nearest objects from the query object. This is called the approximate-similarity search. In this, objects are described in terms of high-dimensional feature vectors, and therefore, the similarity search is conducted by calculating the distance between the query and each searched-object on the underlying space of features [WHCL’06a, WHCL’06b]. Returned objects are then ranked based on that distance metric. That is, for a query object $q$, the goal of any approximate-similarity search technique is to find all objects $s$, such that the distance $\lambda(q, s_i)$ between $q$ and $s_i$ is less than a certain number. In general, the best matching objects evaluate the smallest value for the function $\lambda$. Although it saves the high cost of intensive pair-wise comparisons by limiting such comparisons to a small number of searched-objects, the efficiency and the accuracy of an approximate-similarity search technique depends –respectively- on amount and the type of extracted features [WDJLCL’07]. Fortunately, XML implementations of similarity search techniques have fewer drawbacks as there is a certain number of features to be encoded in order to perform the similarity-check for the majority of XML query classes [ZOIA’06a]. XML indexing techniques using this mechanism are discussed in Section 7.3. The next section explores some data-structures that are used to encode the features of the indexed objects.

7.2. Implementations and Data-structures

Similarity searching is widely used in today’s informational retrieval technology. This section aims to highlight some areas where such techniques are used to show their power. To achieve this goal, two examples of such implementations, namely searching graphs’ databases and text processing, are discussed in particular because they link to the discussion of the XML’s feature-based indexing in terms of the size and the type of the manipulated data. Other implementations include archived video [ROS’04] [WHCL’06a] [WHCL’06b] [GACW’05] techniques. The discussion is intended to be brief for the purpose of introducing the next section (Section 7.3).
**Text Processing:** Similarity search has been used in text processing and document retrieval for a long time [SL’68] [DM’85]. The main purpose of this terminology is to find all documents (text snaps) that match a query which mainly composed of a set of keywords. One of the earliest and simplest techniques was inverted-lists [ZMS’92]. In this, each document (or a piece of text) is inspected for the required keywords after stop-words removal and stemming [FF’03] the remaining ones. Then, the frequency of each keyword is calculated in each document to give the document its weight (or features). Next, this information is stored on what so called inverted-list (usually use hash tables [TS’84]) which stores, for each keyword, all document IDs and the corresponding frequency of the current keyword. By using Robertson’s BM25 formula [RWBGP’95], similarities between the query (which is also represented in a form of keywords and their frequencies) and each document is calculated to give each document its rank against the query. Depending on the ranking results, a document is either returned by the query processor or discarded. In conclusion, there are other similarity-calculation techniques as well as text-processing methodologies. However, listing and discussing such techniques and methodologies are outside the scope of this report. In [MZH’05], Maffot et al. have outlined some IR-related proposals written between 1981 and 2004. Recent IR-related proposals are also found in [SH’06] [ZM’06] [BCG’05] [TNI’04].

In XML context, there are two main occasions where inverted lists can be used effectively. In document-centric XML databases, contents of XML documents play the major role in determining the results returned rather than the document’s structure. Therefore, variations of existing information retrieval (IR) techniques are used to index and query document-centric XML databases. An example of such an implementation is the RANK system [GSBS’03] which differs from any non-XML oriented IR systems in that; 1) it matches deeply-nested XML elements instead of matching the entire document, and 2) ranking process is based on the granularity of XML elements rather than the granularity of the entire document. Detailed discussion and more examples of this type of implementations are found in Section 3.1. Inverted lists are also used to index data-centric XML databases. There are cases where such implementations used to index tag names and virtual nodes that are created to hold leaf-node values in structural indexes. More discussion on this is found in Section 5 and Section 6.

**Graph Indexing:** Similarity search has also been used to search chemical-compounds databases [YYH’05] [FR’05] [YZYH’06]. A chemical compounds is often represented in cycled-graph, and a chemical database is then defined as a forest of such disconnected graphs. This is similar to an XML database which is modelled as a forest of XML trees (see Section 2.2). So, similarity search is used to check for the query’s (a compound graph's) existence in a chemical-graphs database by matching some features of the query with each graph in the database. It important to notice that such a comparison is used to eliminate as many non-matching graphs as possible at an early stage before conducting any pair-wise similarity computation in order to return the final results.

In order to facilitate the similarity filtering, Yan et al [YZYH’06] used what is called feature-graph matrix [SWG’02]. In this matrix, indexed chemical graphs are aligned in the columns whereas each row in the matrix represents a feature. Each entry in the matrix reflects a calculated value of a specific feature for the corresponding graph. For example, if there are four graphs (say $G_1$, $G_2$, $G_3$, and $G_4$) each of them is searched for three features (say $f_a$, $f_b$, and $f_c$), then a corresponding feature-graph matrix is given in Figure 7.1. A very good phenomenon about this representation is that a graph and/or a feature can be easily added or removed to/from the index without reconstructing the index. In addition, the index can be implemented using a linked-list where each entry points to an array (or another liked-list) that contains entries of a row in the matrix [YZYH’06].

$$\begin{pmatrix}
G_1 & G_2 & G_3 & G_4 \\
0 & 2 & 1 & 0 \\
1 & 0 & 1 & 2 \\
0 & 0 & 2 & 3
\end{pmatrix}$$

**Figure 7.1: Example: Feature-graph Matrix**

In the literature, FIX [ZOIA’06a] seems to be the first and the only solution that uses a feature-based matrix technique specifically to index XML data. However, there are several differences between this XML-based index (i.e. FIX) and the one used in [YZYH’06] or any graph-based index. A detailed
A discussion of FIX and its related technologies is the subject of the next section (Section 7.3). More about graph and matrix theories and their possible employment in indexing XML is outlined in Section 8.

7.3. XML Feature-based Indexes

As in chemical-graphs databases, feature-based indices also can be used to speed XML processing [YZHY'06]. However, there are major differences between chemical graphs and XML trees. For example, chemical graphs -- in most cases -- contain cycles [YZHY'06] whereas cycles in XML trees are rare, and if they exist (e.g. using IDREF and IDREFS attributes) they can be avoided or modelled in acyclic forms [ABS'00]. This phenomenon makes XML feature-encoding algorithms easier than those for cycled-graphs. On the other hand, XML trees are much larger, in terms of the number nodes and edges, than chemical graphs. This fact implies a larger number of XML features to be indexed, and consequently more storage space is required to represent these features. For example, when using a feature-based matrix [SWG'02] to encode XML features, matrix rows and columns grow linearly due to the nature of the XML data and its frequent update operations [ZOIA'06a]. Although this is not a problematic issue because such complexity can be reduced by using either a corresponding bi-simulation graph [HHK'95] [PT'87] or an efficient data-structure, issues such as query processing and data updates need more attention.

Based on the above discussion about the requirements of graph feature-based indices in general, and XML feature-based in particular; the three major issues to be considered when designing any XML feature-based index can be summarized as:

a) What XML features need to be encoded so that fast and accurate comparisons can be conducted?
b) What is the cost of the features' encoding process in terms of storage space, computation complexity, and processing time?
c) How can feature-based indexes be updated when the underlying data changes?

This section discusses the only two XML feature-based indexing techniques found in the literature. These two indexes are; a scalable bitmap XML index proposed by Yoon et al [YKKC'02], and the FIX index that is proposed by Zhang et al [ZOIA'06a].

The bitmap index proposed in [YKKC'02] is basically designed to index both the structure and the content of multi-documents XML databases (e.g. Figure 7.2). Having said that an XML database is composed of many XML document, say for example $d_1$, $d_2$, $d_3$, …, $d_n$; the idea is encounter the all possible paths from these documents (say $p_1$, $p_2$, $p_3$, …, $p_m$) and then encode the existence of these paths in each document by in a two-dimensional matrix with binary entries. Each entry in this matrix has a value of “1” if a path is found in the corresponding document or “0” otherwise. Figure 7.3 illustrates this structure. So, for XML queries that do not contain any value comparisons, one can easily identify all matching documents by looking at the column corresponds to the query’s path and retrieve all rows (i.e. documents) which have values of “1”.

To include documents’ contents (including the distinct words in textual values, element names, attribute names, and other texts) in the index; a three-dimensional matrix is used. Similar to encoding the path existence in each document, the new dimension in the bitmap-index matrix represents the existence of the words/tokens (i.e. $t_1$, $t_2$, $t_3$, …, $t_k$). The [YKKC'02] names this new structure as BitCube (See Figure 7.4). Therefore, a BitCube for an XML database consisting of many document is defined as BitCube = $(d, p, t, b)$, where the variable $d$ ranges in the documents number, $p$ denotes to all possible paths, $t$ spans for all words/tokens, and $b$ has a value either “0” or “1”.

Apart from other technical infrastructures of the proposed indexing technique in [YKKC'02], the BitCube representation is associated with several operations which build and query the encoded information. For example, the ePath_Slice takes a path as an input and returns a two-dimensional bitmap matrix which represents a set of documents and words associated with them. Similarly the operation Word_Slice and Document_Project respectively take a word or a document and return a two-dimensional bitmap matrix that contains the other two dimensions of the BitCube.
Although the above index structure appears good and simple in terms of speeding up the XML regular queries (i.e. queries that use regular path expressions), there are several facts that might affect its performance. As is clear from its structure, the BitCube is designed to index a multi-document XML databases. However, the structure of the composing XML documents are distinct; that is a set of related XML documents rarely contain a same vocabulary of element and attribute names, and therefore the set of XML paths to be encoded will become huge. This in turn increases the path’s dimension in the BitCube representation and consequently, the time and space for locating and storing such a large set of paths will become costly. The problem becomes even worse for the “word”’s dimension as the number of words is assumed to be infinite. Also the advantage of indexing multi-documents in a single index is of less-importance as the intersection between the path’s sets of those documents is relatively small. In the case of BitCube representation, a better performance might be obtained by using some hashing functions to cluster the large sets of words and paths but such recommendation was not mentioned in [YKKC’02].

The second feature-based index in [ZOIA’06a] is basically designed to index a single-document XML databases. The index called FIX (stands for Feature-base Indexing technique for XML database) and it uses feature-based matrices [SWG’02] to determine the query pattern in XML trees. The basic idea is to construct two featured matrices; one matrix encodes some features of XML trees, and a similar featured
matrix is used to encode the query pattern features. Then, the algorithm uses mature spectral graphs theorems [L'99] [C’00] to determine whether the XML tree matrix can induce the query matrix.

FIX works as follows. The XML matrix encodes node names (i.e. elements, attributes and could be textual value) and edges relationships (i.e. structural relations such as parent-child) between these nodes. For example; if an XML tree has \( n \) nodes, then its corresponding matrix will be \( m \times m \) matrix, where \( m \) is the set of nodes in a corresponding bi-simulation [HHK’95] [PT’87] graph, so \( m \leq n \). The bi-simulation graph -of an XML tree- is the minimum tree that represents all possible distinct paths of the actual XML tree (Section 5.2). It is used in this approach -instead of the complete XML tree- in order to reduce the size of the matrix representation since it is well-known that there is a finite-set of distinct paths for every XML tree while the XML tree may contain a huge number of redundant paths. Therefore, FIX is a “pruning index that can be built on the top of any existing XPath query processor to achieve better query performance” [ZOIA’06a]. In other words, FIX approach only checks the existential relationship of a query pattern into an XML tree, the actual results are returned via an additional process. As an example, a bi-simulation graph for an XML tree shown in Figure 7.5 is given in Figure 7.6 while Figure 7.7 illustrates the corresponding feature-matrix based on the FIX algorithm. Details of how the entries of the matrix are calculated can be found in [ZOIA’06a] and the underlying technical report [ZOIA’06b].

Textual contents in XML documents are not directly supported by the FIX technique. It is clearly stated in [ZOIA’06a] that, FIX is concerned only with handling a subset of twig-patterns where there is no use of predicates and value comparisons. However, to overcome this limitation, each textual value is treated as a value-node. The tag name for a value-node is calculated using a simple hashing algorithm and the corresponding hash-key is added to the bi-simulation. Unfortunately, this enlarges the node-set in the bi-simulation graph and consequently increases the storage space as well as slows the query evaluation process.

Updates are also a bottleneck of FIX approach. In general, XML documents are subject to frequent updates (e.g. inserting, deleting, and moving nodes). Obviously these updates alter the underlying XML tree structure and consequently its bi-simulation graph(s) which causes re-calculation of the entries of the corresponding feature-matrix.

![Figure 7.5: A bibliography XML Database](image1)

Adapted and modified from [ZOIA’06a]

![Figure 7.6: A Bi-simulation Graph of the XML tree in Figure 7.5](image2)


<table>
<thead>
<tr>
<th>Edge</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>[bib,article]</td>
<td>1</td>
</tr>
<tr>
<td>[article,author]</td>
<td>2</td>
</tr>
<tr>
<td>[bib,book]</td>
<td>3</td>
</tr>
<tr>
<td>[book,author]</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[article,title]</td>
<td>11</td>
</tr>
<tr>
<td>[book,title]</td>
<td>12</td>
</tr>
</tbody>
</table>

\[
M = \begin{pmatrix}
0 & 2 & \cdots & 12 \\
-2 & 0 & \cdots & 0 \\
\cdots & \cdots & \cdots & \cdots \\
-12 & 0 & \cdots & 0
\end{pmatrix}
\]

Figure 7.7: Matrix Computations of the Bi-simulation Graph in Figure 7.6

7.4. Conclusion

Similarity search has proven its strength and efficiency in text processing via the use of various IR techniques. However, more effort is required to apply such techniques in the XML context. Although there have been several attempts to employing IR technology in querying and indexing XML data, most of these proposals were based on document-centric databases rather than data-centric documents. In addition, those attempts lack most of IR-related functionalities such as document’s weighing, ranking and relevance retrieval [See Section 3.1].

In terms of feature-based searching, very little attention has been given to indexing XML data using such techniques. There are two proposals; the first technique indexes a set of XML documents in a bitmap-based matrix while the second encodes a corresponding bi-simulation graph of an XML tree in a matrix that consists of arbitrary unique numbers for each edge of the bi-simulation graph. Both techniques then use the theories of matrices along with some similarity searching techniques to evaluate XML queries. However, among other symptoms, these techniques seem to suffer from the same XML update problem that other XML indexes do because of their infrastructures (e.g. the use of bi-simulation graph in [ZOIA’06a]). Logically, basing such indexes on the entire XML tree and using a proper XML tree representation with an efficient compressing algorithm may leads to a better query performance and update maintenance. The research, which is motivated and proposed in the next section, is based on this hypothesis.
8. Motivations and a Research Proposal

This section presents the motivation for studying XML databases in general and their indexing techniques in particular. In addition, the three main problems that are found in the existing XML indexing techniques are summarized. As result of these indexing problems, a proposal for constructing a new XML index that may reduce or eliminated the effects of such problems is put forward.

The section is structured as follows: Section 8.1 divides the research motivation between expressing the importance of studying the XML databases and showing the strengths of the feature-based index in the XML context. In addition, the section outlines the idea of a new feature-based index which is hypothesised to address three major problems in the existing XML index. These problems are revisited from previous sections of this review and highlighted in Section 8.2.

8.1. Motivations

This research is initially motivated by the importance of XML importance in the today’s database management technology. So, the first subsection (of this section) revisits some XML’s features that attracted database community to research the notion of XML databases. The research is also motivated by feature-based indexes’ strengths in the evaluation of XML’s regular path queries especially for the large XML databases. This motivation is presented in the second subsection. The third and the fourth subsections respectively describe and exemplify the structure of a new feature-based index that is hypothesised to addresses some problems in the existing XML indexes.

8.1.1. Studying XML Databases

Sections 1 and 2 discussed the features of the XML and its data model which made it a widely accepted technology in the today’s information technology era. Revisiting such features, the XML is a language which:

- has a simple syntax: it consists of a serious of opening and closing tags that describe the data as well as the structure,
- is portable: that is, it is not tied to a specific implementation because it is encoded in Unicode textual format,
- is extensible: its infinite vocabulary (i.e. tags and attribute names) allows the XML an ability to represents data from other models such as the relational and the object-oriented data models, and
- is readable by both humans and machines

These valuable features have made the XML a standard medium for transferring data over the Web as well as exchanging information between the heterogeneous systems [W3C1]. As a result, the amount of the data stored in the XML’s repositories, and the number of XML’s users has increased dramatically and therefore, important database issues, such efficient indexing techniques, have become critical requirements.

In Section 4, 5, 6 and 7, different types of XML indexes were discussed comparing their strengths and shortcomings in indexing XML databases of different sizes. Among these XML indexes, the feature-based XML approaches seem to be a promising technology in indexing XML data efficiently for a wide range of XML queries that are represented by regular path queries (e.g. XPath2.0 queries and a subset of XQuery1.0 queries as the XQuery1.0 query language is built on the top of XPath2.0 query language). This is discussed further in the next section.
8.1.2. Motivating Feature-based XML Indexing

As it is well-known, there is no clear division between data and schema definitions in the context of XML databases. Therefore, getting a desired set of data requires a proper navigational algorithm over the document’s structure starting from the top-level element (i.e. the root of the corresponding XML tree) downward to the leaf nodes which contains the actual data. Reaching a specific point in the XML tree requires knowledge of a certain set of information about the XML tree at each branch while walking through its paths. In widely accepted XML query languages such as XPath2.0 and a subset of XQuery1.0 query languages, this information is normally expressed in a form of path axes (or XML elements relationships), for example child axis, self-or-descendant axis, and so no (See [W3C4] for other XPath’s axes). By treating such elements’ relationships as mid-way features or mile-stones while navigating through both the XML tree paths and/or the query expression paths, the process of selecting specific data from an XML database through an XPath query then becomes a matter of following these mile-stones until reaching the desired data. This is often referred as “similarity search” discussed in Section 7.1.

It has been shown in the literature (e.g. [TVBSSZ’02] [LLCC’05] [CKM’02]) that regular path queries can be easily answered by encoding only two structural relationships between the elements in an XML database. These relationships are the parent-child and the ancestor-descendant. In addition, other structural relationships such as following and preceding axes can be derived from them. Therefore, encoding the two basic relationships as features in what is so called a feature-based matrix, and using a proper similarity-search algorithm to compare the query’s feature with the XML database’s features might lead to a better XML indexing. The purpose of this research is then to formalise, design, and possibly, to implement and evaluate a new feature-based index which encodes the elements’ structural relationships in a set of feature-based matrices.

The next subsection (Section 8.1.3) describes the structure of a new feature-based index that may answer the index problems discussed in Section 8.2.

8.1.3. The Index Idea

a) Background and Advantages:

The basic idea of the new feature-based XML index is to speed-up the process of evaluating a wide class of XML queries that are represented by regular path queries. The following features of the proposed index contribute the index's efficiency:

- Conduct both the query evaluation phase and output construction phase by consulting the index representation only. This eliminates the time and the effort required to switch between the index representation and the actual database file in order to construct the query’s result.
- Allow the index to reside in the computer’s main memory (by compressing the index) while processing the underlying XML database. Again this is going to save the query processing time by eliminating the expensive disk-based I/O operations.
- Employ the power of the feature-based approaches (described in Section 8.1.2) in answering regular path XML queries which can be evaluated by comparing and contrasting a set of XML feature for both; the XML database and the regular path query.
- Employ a novel representation for XML data using feature-based matrices to represent the XML document’s structural relationships which are required to answer regular path queries as described in Section 8.1.2.
- Allow different levels of the index’s compression necessary to keep the index in memory. These include:
  - Clustering the feature-based matrices into a single feature-based matrix
  - Employing an effective sparse-matrix compression technique from matrix and graph theories to further compress the clustered feature-based matrix.

This fact should reduce the size of the index so that it fits in the computer’s main memory and therefore uses memory-based query’s evaluation transactions.
• Allow cheap and systematic index’s update operations that are offered by the novel XML data representation mentioned above. These include:
  o Node-Insert: which can be done by adding a column and a row for the new node in the feature-based matrices (or the clustered feature-based matrix)
  o Node-Delete: which can be done by removing the corresponding column and row from the feature-based matrices (or the clustered feature-based matrix)
  o Label-Update: which has no direct effects on the feature-based matrices because the node-set names (i.e. labels) are kept in a separate list, and any node-label in that list can be reached efficiently using an existing list-searching technique such as Binary-search trees or Hash-tables.
  o Node-Shift: to be investigated for possible automation.

In addition to the above node-update operations, twig-update operations will be investigated for similar automation.

b) Matrices Construction:

Having the above facts in mind, the whole idea seems to be concentrated around how the XML’s structural relationships are represented in a set of feature-based matrices. To achieve such a presentation, necessary structural relationships for the evaluation of each class of regular path queries are identified and each of them is encoded as binary entries in separate feature-based matrix. Each feature-based matrix is constructed as following:

• The matrix’s dimension is n rows by n columns, where n is the number of nodes in the indexed XML tree. So, the header of each row and column is a unique identifier corresponds to a node in the XML tree.

• Each matrix $M_i$ represents a certain structural relationship $R$, where $R$ belongs to the set of existing structural relationships in the XML tree (i.e. Parent, Child, Ancestor ...etc). So, for example there is a matrix $M_i$ that encodes the parent-relationship between each pair of the tree nodes.

• A matrix’s entry $M_R(i,j)$ is either 0 or 1. If the $R$-relationship is exist between the $i^{th}$ node and the $j^{th}$ node, then $M_R(i,j) = 1$, otherwise $M_R(i,j) = 0$. For example, if the node $a$ is the parent of the node $b$, then the Parent-relationship matrix contains the following entry $M_P(a,b) = 1$, where $a_i$ is the row number corresponds to the node $a$, and the $b_i$ is the column number corresponds to the node $b$.

c) Query Evaluation:

By constructing several feature-based matrices using the above algorithm, evaluating a regular path XML query becomes a matter of consulting a subset of these matrices depending on the query’s steps. For example, the XPath query $\langle a \rangle \langle b \rangle$ can be evaluated by accessing the Parent-Matrix and the Descendant matrix respectively for each step of the query. A query-processor that uses the new feature-based index will be constructed in later stages of this research.

d) Storage Space Complexity:

In terms of storage complexity, at the first glance the index seems to require massive storage space which can be approximated by $O(m) \times O(n^2)$, where $n$ is the number of nodes in the XML tree and $m$ is the number of encoded features (i.e. the number of matrices). However, the $m$ complexity can be eliminated to $O(1)$ in two steps:

• Some feature-based matrices can be obtained by applying several geometric transformations. For example, the Parent-Matrix can be calculated from the Child-Matrix as follows:

\[
M_P(i,j) = \text{Reflect} \{ \text{Rotate}_y \{ M_C(i,j) \} \},
\]

similarly, the Ancestor-Matrix can be calculated from the Descendant-Matrix as follows:

\[
M_A(i,j) = \text{Reflect} \{ \text{Rotate}_y \{ M_D(i,j) \} \}, (\text{the proof is delayed})
\]
• Next, the resulting feature-based matrices (say $\tilde{M}_1$, $\tilde{M}_2$, ..., $\tilde{M}_k$) are clustered in a single matrix ($M_{all}$) as following:

$$M_{all}(i,j) = \text{map}(\tilde{M}_1(i,j) \oplus \tilde{M}_2(i,j) \oplus \ldots \oplus \tilde{M}_k(i,j))$$

Where $\oplus$ is a string concatenation operator, and the $\text{map}(x)$ function produces a single-character map for the concatenated string $x$.

For example, if $k=4$ then a possible value-set for the function $\text{map}(x)$ is the hexadecimal numeric notations (i.e. $\text{map}(x) \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$)

On the other hand, because most of the master feature-matrix $M_{all}$ are zeros, the $O(n^2)$ complexity can be reduced by applying an effective sparse-matrix compression technique. This issue will be investigated in the coming stages of the research.

![Figure 8.1: A Framework Structure for the New Feature-based Index](image)

**e) Updates Computation:**

In terms of the index update computation, the effect of the two basic update operations were already discussed above in this section. While the Node-Insert and the Node-Delete operations cost nothing except adding and removing the corresponding row and column to/from the feature-based matrices (apart from the cost of the un-map() and map() functions), the effect of inserting a twig (twig-Insert) and removing a twig (Twig-Delete) are to be investigated. One way to conduct Twig-Insert and Twig-Delete operations is to model them in a series of Node-Insert and Node-Delete respectively. In addition, the Node-Shift and Twig-Shift operations can also be modelled as a series of Node-Insert and Node-Delete operations. These issues are subject to investigation in further stages of the research.

The basic framework for constructing the index and using this index to query the underlying XML database is illustrated in Figure 8.1. The following section presents an example to show the simplicity of
the index’s basic structure which can be considered as another motivation towards proposing the new feature-based index.

8.1.4. A Motivating Example

To show how the XML’s structural features can be encoded in feature-based matrices, Figure 8.2 shows an XML tree representing a simple XML database. Figures 8.3, 8.4, 8.5 and 8.6 draw four feature-based matrices that respectively encode the parent-, child-, ancestor- and descendant relationships. For example, from Figure 8.3, we can get all the child-nodes of the node “c” by selecting all the rows that contain a value of “1” under the “c” column. So, the resulting node-set contains the nodes “f” and “g”. Therefore, the basic idea of the proposed index is to use these feature-based matrices in order to select the desired nodes at each step of an XPath query as described in Section 8.1.3 and illustrated in Figure 8.1. The rest of the idea will be developed progressively during this research.

![An XML Tree](image)

**Figure 8.2: An XML Tree**

<table>
<thead>
<tr>
<th>Parent-of</th>
<th>Child-of</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Matrix Image" /></td>
<td><img src="image" alt="Matrix Image" /></td>
</tr>
</tbody>
</table>

**Figure 8.3: Parent-of Relationship Matrix of Figure 8.2**  
**Figure 8.4: Child-of Relationship Matrix of Figure 8.2**

<table>
<thead>
<tr>
<th>Ancestor-of</th>
<th>Descendant-of</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Matrix Image" /></td>
<td><img src="image" alt="Matrix Image" /></td>
</tr>
</tbody>
</table>

**Figure 8.5: Ancestor-of Relationship Matrix of Figure 8.2**  
**Figure 8.6: Descendant-of Relationship Matrix of Figure 8.2**
The four feature based matrix (in Figures 8.3, 8.4, 8.5 and 8.6) can be clustered in a single feature-based matrix using a simple mapping function that map the concatenated binary codes to hexadecimal values. The resulted feature-based matrix is given in Figure 8.7.

\[
\bar{M} = \begin{pmatrix}
0 & C & 0 & C & 4 & 0 & 0 & 3 \\
3 & 0 & 0 & 0 & C & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & C & 3 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 3 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 3 & 0 & 0 & 0 & 0 & 1 \\
C & 4 & C & 4 & 4 & C & 4 & 0 \\
\end{pmatrix}
\]

Figure 8.7: The Master Feature-based Matrix for Figure 8.2

8.2. Indexing Problems in Existing Approaches

Most of the existing indexing techniques from the all classes discussed in Sections 4, 5, 6 and 7 share three major problems that affect both the cost of the index and its performance. These problems are: 1) the index size, 2) the index computational cost, and 3) the index updatability. Each of these problems is revisited in the following subsection respectively, and possible solutions (using the proposed feature-based index) are hypothesised along with that discussion.

8.2.1. The Index Size Problem

Most of the discussed indexing techniques in this review lack the scalability feature -that is determined by the index size- in one form or another. While some indexes are basically designed to handle a single-document database, which is far from the reality; others are designed to be memory-based indexes and therefore, large-scale XML documents become a bottle-neck for such indexes. For those approaches which use a disk-based indexing techniques (e.g. [WJWLLL’05]) to index the large-scaled XML databases, the scalability problem results in a trade-off between the capacity of the index and its performance in terms of the index-processing time that is required to locate the desired disk pages.

Although the physical storage space is not a problematic issue nowadays, minimizing the index size is an important aspect for a memory-based index. The proposed feature-based index should reside in the computer’s memory and all its operations are to be conducted on the memory version of the index. So, based on these assumptions, the index size can be compacted by using one or more of the following techniques:

- The corresponding bi-simulation [HHK’95] [PT’87] version of the XML database has a major effect in reducing the actual XML tree. Encoding the bi-simulated version of the underlying XML database in the proposed feature-based matrices will result in further reduction in the index size
- It can be easily proven that some of the above feature-based matrices can be obtained by mathematically transforming other feature-based matrices. For example, the child-relation matrix can be calculated from rotating the parent-relationship matrix by 90° and then reflecting the rotated matrix around y-axis. This phenomena allows the index to cluster several feature-based matrices in a single matrix and consequently reduce the required storage space for the index
• Using certain data-structure representation and/or one of the existing sparse-matrix 5 compression techniques will lead to a further index compression due to the nature of the structural relationships that compose XML trees.

For the above three aspects, there are existing algorithms from the matrix-algebra and the computerized-data structure fields which can be directly employed in the intended feature-based indexing technique. The selection of a suitable technique will be investigated during the index’s design process. The next subsection presents the second common problem in the existing XML indexes.

8.2.2. The Computation Cost Problem

Relational-based XML indexes discussed in Section 5.1 use node-labelling algorithms to encode the XML’s structural relationships. One deficiency of this implementation is the need for relatively-complex computations to calculate the elements’ relationship during the query evaluation. Furthermore, accessing these indexes is not enough to evaluate the answer to a query without consulting the actual database file. In the new feature-based representation, the answers of a structured-based query can be directly obtained from the index file itself without the need to access the actual XML document.

In the native XML indexes such as sequenced-based techniques, the high computational cost results from the following aspects:
• the way the XML data sequences are formed and encoded (i.e. the algorithms used to encode and build the database textual-stream)
• the way to conduct the comparison process between the query’s sequence and the document’s sequence

The same computational complexities are found in the existing feature-based indexes. Furthermore, in both the existing sequence- and the existing feature-based techniques, a very high computation cost is incurred during the re-construction process of the index when the underlying database changes. This results in the index-updatability problem which is discussed in the following section.

In the proposed feature-based index, all feature-based matrices can be constructed by parsing the XML documents only once. Furthermore, the data-selection process (i.e. the output construction) is a matter of excluding those entries which have zero values in a corresponding feature-based matrix.

Further complexity reduction in the proposed feature-based index can be demonstrated during the update operation which is discussed below.

8.2.3. The Index Update Problem

This is a common problem in all indexing classes that are discussed in this report. Relational approaches discussed in Section 4 and 5.1 experience expensive update operations when the underlying XML data changes. The update problem results from; 1) updating the label of the XML nodes for those techniques which rely on the node’s labelling to represent the structural information of the XML document [TVBSSZ’02] [SLFW’05] [ZNDDL’01], and/or 2) updating the paths’ list for those techniques which store the all possible paths found in the XML document in order to use them for the structural-queries evaluation process [CLO’03] [CMS’02]. The update problem is even worse in the sequence-based approach (Section 6) where the update has to occur in several positions in the textual stream that encodes the underlying XML database.

With respect to the existing feature-based indexes, the two techniques discussed in Section 7 suffer from the updatability problem as well. In the [YKKC’02]’s proposal, the problem is very clear because the technique encodes all possible ePaths found in the underlying XML database and uses them to construct the feature matrix (See Section 7.4). As a consequence of this, any simple change in the document’s structure (e.g. deleting a non-leaf node) could lead to many changes in the ePath set especially when the

---

5 In the mathematical subfield of numerical analysis a sparse matrix is a matrix populated primarily with zeros
change happen towards the root of the XML tree. Therefore the underlying feature-based matrix must reflect this change by updating many columns that correspond to a set of ePaths affected by the change. This problem is more or less same in the FIX index [ZOA’06a]; the second feature-based index discussed in Section 7.4, because the FIX algorithm simply assign sequential numbers to the whole existing edges between the nodes’ pair in the bi-simulation graph. These numbers are aligned in the feature-based matrix; and for those un-related pairs, they are assigned zero values.

In the proposed feature-based index, the two basic update operations (i.e. the node-insert and the node-delete) are straightforward. For example, to inset the leaf-node “i” in Figure 8.2, it will be sufficient to add a column and a row at the end of the feature-based matrix that reflect the corresponding structural relationships between the inserted node and the existing nodes. The resulting feature-based matrices after inserting the leaf-node “i” are shown in Figures 8.8, 8.9, 8.10 and 8.11 (the shaded columns and rows encodes the “i”th node structural relationships). Similarly, when a node is to be deleted from the XML tree, it will be sufficient to remove the corresponding column and row from the feature-based matrix. Other update operation can be modelled by conducting a series of node-insert and node-delete operations.

<table>
<thead>
<tr>
<th>Parent-of</th>
<th>Child-of</th>
</tr>
</thead>
<tbody>
<tr>
<td>c f b g h d e a i</td>
<td>c d 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>f 1 0 0 0 0 0 0 0 0 1 0</td>
<td>f d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>b 0 0 0 0 0 0 0 0 0 1 0</td>
<td>b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>g 1 0 0 0 0 0 0 0 0 1 0</td>
<td>g h 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>h 0 1 0 0 0 0 0 0 0 1 0</td>
<td>h d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>d 0 0 0 0 0 0 0 0 0 1 0</td>
<td>d e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>e 0 0 1 0 0 0 0 0 0 1 0</td>
<td>e a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>a 0 0 0 0 0 0 0 0 0 1 0</td>
<td>a i 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>i 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.8: Parent-of Relationship Matrix of Figure 8.2 (after inserting i)

<table>
<thead>
<tr>
<th>Child-of</th>
<th>Parent-of</th>
</tr>
</thead>
<tbody>
<tr>
<td>c f b g h d e a i</td>
<td>c d 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>f 0 0 0 0 0 0 0 0 0 1 0</td>
<td>f d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>b 0 0 0 0 0 0 0 0 0 1 0</td>
<td>b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>g 1 0 0 0 0 0 0 0 0 1 0</td>
<td>g h 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>h 0 1 0 0 0 0 0 0 0 1 0</td>
<td>h d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>d 0 0 0 0 0 0 0 0 0 1 0</td>
<td>d e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>e 0 0 1 0 0 0 0 0 0 1 0</td>
<td>e a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>a 0 0 0 0 0 0 0 0 0 1 0</td>
<td>a i 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>i 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.9: Child-of Relationship Matrix of Figure 8.2 (after inserting i)

<table>
<thead>
<tr>
<th>Ancestor-of</th>
<th>Descendant-of</th>
</tr>
</thead>
<tbody>
<tr>
<td>c f b g h d e a i</td>
<td>c d 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>f 1 0 0 0 0 0 0 0 0 1 0</td>
<td>f d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>b 0 0 0 0 0 0 0 0 0 1 0</td>
<td>b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>g 1 0 0 0 0 0 0 0 0 1 0</td>
<td>g h 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>h 0 1 0 0 0 0 0 0 0 1 0</td>
<td>h d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>d 0 0 0 0 0 0 0 0 0 1 0</td>
<td>d e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>e 0 0 1 0 0 0 0 0 0 1 0</td>
<td>e a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>a 0 0 0 0 0 0 0 0 0 1 0</td>
<td>a i 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>i 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.10: Ancestor-of Relationship Matrix of Figure 8.2 (after inserting i)

<table>
<thead>
<tr>
<th>Descendant-of</th>
<th>Ancestor-of</th>
</tr>
</thead>
<tbody>
<tr>
<td>c f b g h d e a i</td>
<td>c d 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>f 0 0 0 0 1 0 0 0 0 0</td>
<td>f d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>b 0 0 0 0 0 0 0 0 0 1 0</td>
<td>b e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>g 0 0 0 0 0 0 0 0 0 1 0</td>
<td>g h 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>h 0 0 0 0 0 0 0 0 0 1 0</td>
<td>h d 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>d 0 0 0 0 0 0 0 0 0 1 0</td>
<td>d e 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>e 0 0 0 0 0 0 0 0 0 1 0</td>
<td>e a 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>a 1 1 1 1 1 1 1 1 1 1 0 1</td>
<td>a i 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>i 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.11: Descendant-of Relationship Matrix of Figure 8.2 (after inserting i)

8.3. Conclusion

This section (Section 8) has highlighted three major problems that affect most of the existing XML indexing techniques. Using the index structure discussed in Section 8.1.3, this research tries to formalize, and design an alternative feature-based index which may results in better query performance.
9. Conclusion

This article discussed the problem of indexing XML databases. Because of its popularity as a standard medium for transferring the information over the Web as well as exchanging data between heterogeneous systems; the size, the users, and the distribution of XML databases have dramatically increased during the last decade. Therefore, it becomes crucial to investigate the performance of the existing query optimization and indexing techniques for XML databases.

This article initially introduced some basic requirements for XML’s indexing implementations. Several approaches to querying content-based XML databases efficiently were discussed. Most of the existing approaches merge techniques from both the database community and the information retrieval technology which lead to a better query performance than applying either of them separately. It has been noticed that, the most critical problem that might be faced during the querying of document-centric databases is the process of constructing the most meaningful XML fragment that to be returned by an XML query.

The article also discussed various techniques that have been proposed to minimize the index size in order to allow the index to reside in the computer’s memory (and/or consume less storage space) and therefore gain a faster query evaluation by reducing the expensive I/O disk operation. The discussion shows a trade-off between the index size and its applicability; that is as more compression techniques are applied to the XML database, as fewer query classes can be evaluated using the compressed version of the database. Among the discussed compression approaches, the summary graphs –of an XML tree- is the best compression technique in that it reduces the index size, satisfies a wider range of XML queries, and performs fewer computations when evaluating XML path queries.

The rest of this article grouped the existing structure-based XML indexes according to the methodology used to encode the XML’s paths. Relational database management systems offer a good opportunity for indexing XML databases as they use the RDBMS’s mature indexing technology for such purpose. However, this implementation is tied to specific platforms and this is incompatible with the popularity and simplicity that were intended for the XML data model. On the other hand, native XML indexes use three different methodologies to encode the XML’s structural relationships (e.g. parent-child and ancestor-descendant relationships). These are path-summaries, sequence-based approaches, and feature-based approaches. Among these, the feature-based approach seems to be the best medium for indexing XML databases because simply navigating through an XML tree is a matter of hunting some guiding features throughout the tree’s structure (and hence the proposed feature-based index in Section 8).

The literature review culminates in the introduction of an idea for indexing XML databases using a novel feature-based index. The index aims to address several problems that encountered earlier. These problems are; large index-size, high computation complexity, and expensive update operations. To achieve a solution to these problems, the index basically encodes the document’s structure (that is represented by the elements’ relationships such as parent-child and ancestor-descendant relationships) into a number of sparse matrices. Each matrix (which is called a feature-based matrix) encodes a specific one-to-one relationship (say for example parent-relationship matrix) between any node-pairs in the XML tree by representing all the nodes as columns and rows in the corresponding matrix and then assign either “1” if the structural-relationship is exist between the node-pair or “0” otherwise. The constructed feature-based matrices are then merged in one feature-based matrix and further matrix compression is obtained by using an existing compression technique from the matrices/graphs literature. The novelty of the new feature-based index is also presented in the methodology used to evaluate XML structured queries using these matrices, and in the way the index is updated so that the comparison computation is reduced and the cost of updating the index is minimized. Both the query evaluation process and the index updating process were outlined in Sections 8.2.2 and 8.2.3 respectively.

Finally, the work conducted so far is a part of an ongoing research on XML databases at the department of Computer Science in the Sheffield University. Preliminary results will be available in a few months since this publication.
References:


Appendix A: Abbreviations List

The following list contains a quick reference for some abbreviated words/phrases that are used in this article:

API an Application Programming Interface
CAM Compressed Accessibility Model
DBA Database Administrator
DOM Data Object Model
DTD Document Type Definition language
FLWOR an XQuery's syntax stands for For-Let-Where-Order By-Return
GML Generalized Mark-up Language
HTML Hyper-Text Mark-up Language
I/O Input/Output (e.g. disk I/O operations)
IR Information Retrieval
KWS Keyword Search
RDBMS Relational Database Management System(s)
SGML Standard Generalized Mark-up Language
SQL Structured Query Language
W3C World Wide Web Consortium
XML The Extensible Mark-up Language
XPath XPath query language
XQuery XQuery query language
XSD XML Schema Definition language