INDEX STRUCTURES FOR XML DATABASES

by

Samir A. Mohammad

A thesis submitted to the
School of Computing
In conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
(March, 2011)

Abstract

Extensible Markup Language (XML) is a de facto standard for data exchange in the World Wide Web. Indexing plays a key role in improving the execution of XML queries over that data. In this thesis we discuss the three main categories of indexes proposed in the literature to handle the XML semistructured data model, and identify limitations and open problems related to these indexing schemes. Based on our findings, we propose two novel XML index structures to overcome most of these limitations: a native index structure called Level-based Tree Index for XML databases (LTIX) and a relational index structure called Universal Index Structure for XML data (UISX).

A proper labeling scheme is an essential part of a well-built XML index structure. We found that existing labeling schemes are not suitable for our index structures and therefore propose a novel labeling scheme, Level-based Labeling Scheme (LLS), which has the advantages of most popular types of labeling schemes while eliminating the main disadvantages. We then combine our LLS labeling scheme with our index structures. An evaluation shows that LLS performs well in comparison to existing labeling schemes using different mappings to relational tables.

We propose the LTIX to minimize the number of joins and matches required to evaluate twig queries, and also to facilitate effective query optimization through early pruning of the space search. Our experimental results show that this approach performs well in comparison to existing state-of-the-art approaches.

We propose the UISX to overcome the key problem with the state-of-the-art approaches, namely that they cannot support efficient processing of twig queries without requiring significant storage. We use a light-weight native XML engine on top of an SQL engine to perform the optimization related to the structure of the XML data prior to shredding. Experimental results show that our approach achieves lower response times than other similar approaches while using less space to store XML data.

Co-Authors

Mohammad, S., and Martin, P., 2009. Index structures for XML Databases. In Li, C., and Ling, T. W. (Eds.). *Advanced Applications and Structures in XML processing: Label Streams, Semantics Utilization and Data Query Technologies*. IGI Global, pp. 98-124, is based on Chapter 2.

Mohammad, S., and Martin, P., 2009. XML Structural Indexes (*Technical Report No. 2009-560*). Kingston, Ontario, Canada: Queen's University, is based on Chapter 2.

Mohammad, S., and Martin, P., 2010. LLS: A Level-based Labeling Scheme for XML Databases. In *Proc. of CASCON 2010*, Toronto, Canada, pp. 115-127, is based on Chapter 3.

Mohammad, S., and Martin, P., 2010. LTIX: A Compact Level-based Tree to Index XML Databases. In *Proc. of International Database Engineering and Applications Symposium*, Montreal, Canada, pp. 21-25, is based on Chapter 4.

Mohammad, S., and Martin, P., 2010. LTIX: A Compact Level-based Tree to Index XML Databases. (*Technical Report No. 2010-570*). Kingston, Ontario, Canada: Queen's University, is based on Chapter 4.

Mohammad, S., Martin, P., and Powley, W., 2010. Relational Universal Index Structure for Evaluating XML Twig Queries. (*Technical Report No. 2010-576*). Kingston, Ontario, Canada: Queen's University, is based on Chapter 5.

Mohammad, S., Martin, P., and Powley, W., 2011. Relational Universal Index Structure for Evaluating XML Twig Queries. Accepted for publication in the *Proc. of the International Conference on Communications and Information Technology – ICCIT 2011*, Aqaba, Jordan., is based on Chapter 5.

Acknowledgements

First and foremost I want to thank my advisor, Dr. Patrick Martin, whose encouragement, guidance, and support along the way enabled me to develop an understanding of the subject and to improve on it. It is an honor to work with him. I would like to thank Dr. Ashraf Aboulnaga for his valuable advice on XML indexing. My sincere thanks to Dr. Selim Akl, Dr. Hossam Hassanein, and Dr. Kathryn Brohman for their important comments and support. I want to express my gratitude to Research Associate Wendy Powley for her friendly support and for listening to me whenever I was excited about a new idea. I thank the members of the database laboratory for being so nice and helpful, also I thank my colleagues and the staff in the School of Computing for providing a pleasant atmosphere to pursue my research.

This thesis would not have been possible without my wife Rola's sacrifice and support. I thank my children Ayah, Mohammad, Sarah, and Abdulkareem for giving me the strength to endeavor every time I felt down in this long journey. I am blessed to have the greatest parents whose prayers, love, and support helped me to hold on to my dreams. My thanks go to my brothers and sisters, especially brother Basheir who provided me with my first PC.

I would like to thank the Natural Science and Engineering Research Council of Canada (NSERC) for funding my research, and Canada for giving me this opportunity and for providing a new welcoming home for my family. Last but not the least, I am grateful to the almighty God for answering my prayers.

Statement of Originality

I, Samir A. Mohammad, certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

(March, 2011)

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ADG	Approximate DataGuide
API	Application Program Interfaces
DAG	Directed Acyclic Graph
DBLP	Digital Bibliography and Library Project
DBMS	Database Management System
DOM	Document Object Model
DTD	Document Type Declaration
ID	Identification
IDREF	Indetification Reference
I/O	Input/Output
IR	Information Retreival
ISO	International Organization for Standarization
LLS	Level-based Labeling Scheme
LTIX	Level-based Tree to Index XML
MPMGJN	Multiple Predicate MerGe JoiN
PRIX	PRufer sequences for Indexing XML
RDBMS	Relational Database Management System
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
UISX	Universal Index Structure for XML
ViST	Virtual Suffix Tree
W3C	World Wide Web Consortium
XML	Exensible Markup Language

Chapter 1

Introduction

Extensible Markup Language (XML) is becoming the dominant method of exchanging data over the Internet. It was endorsed as a W3C recommendation in 1998 [14]. Its roots go back to SGML (Standard Generalized Markup Language) [14]. SGML is an international standard since 1986 (ISO 8879). SGML is a meta-language, that is, it can be used to create new languages in order to describe any kind of data. The differences between SGML and XML arise from the aim to develop a meta-language especially for the needs of the Web and to promote the fast establishment of this language on the Web [117]. XML's implementation, for example, is much simpler than that of SGML and a DTD (Document Type Declaration) does not have to be used with XML documents.

A DTD is used to specify some restrictions on XML data such as, among other things, the relationship between elements and types of elements [14]. XML Schema [121] is an extension to DTD and has been supplied with many features to overcome some of the limitations of DTDs [21]. Both DTD and XML Schema are analogous to a schema in Relational Database Management Systems (RDBMS). Even with the presence of a DTD or an XML Schema, XML data is considered as semistructured [132]. This is due to the possible use of the "any" *Type* of contents in DTD and the <any> *Element* in XML Schema, both of which extend an XML document with arbitrary elements [21] [70] [121].

XML database systems, including the query optimization engine, do not have the advantage of being founded on several decades of scientific research as do relational DBMSs. In contrast to query optimization in relational databases, XML query optimization is a comparatively new research area.

Indexing and querying XML data are active research areas [6] [56] [65] [89] [104] [130] [146]. Methods and techniques from other areas of Information Technology have been adapted to index XML data. Inverted files are used for text-dense XML documents [58] [59] [74] [109] [133] [137]. A Suffix Tree is used by Wang et al. [130] to develop dynamic indexes, and by Zuopeng et al. [147] to build an XML index structure. The Index Fabric by Cooper et al. [37] and the PT index by Li et al. [79] are based on the Patricia (Practical Algorithm To Retrieve Information Coded In Alphanumeric) Trie [73], which is a string indexing scheme. The research on optimization of path expressions in object-oriented database systems [52] and graph-based semistructured data models [1] [3], have been adapted by McHugh and Widom [88] to develop Lore (an XML database management system).

It is worth mentioning here that some researchers emphasize the fact that database technology has to be integrated with Information Retrieval (IR) technology in order to effectively manage XML data [9] [11] [84]. IR technology can be used to handle the unstructured text contents of XML documents, while the database technology can be used to handle the structured part of XML documents.

Many systems have been proposed in the academic and commercial fields to provide either a query engine for XML data or a complete XML database management system. For example, some systems are designed to handle semistructured data [19] in general, including XML documents [3] [15] [20] [46]. Other systems are designed specifically for XML data [12] [26] [38] [47] [102] [112] [128], or have migrated to a fully XML-based data model [53] [87] [88]. Finally, there are languages that are designed to query only XML data [16] [17] [24] [39] [98] [105] [106] [122]. The anatomy of native XML databases is discussed by Feinberg [45].

There are two methods for storing and querying XML databases. The first method maintains the native hierarchical nesting structure of XML databases and is referred to as the *Native Method*. The other method leverages the existing power of RDBMSs that has been established over several decades, and is called the *Relational Method*. Structural indexes play a main role in data retrieval and manipulation in both methods [124]. Both methods have drawn the attention of the research community. As is always the case with indexes, there is a tradeoff between the size and the power of indexes [80] [144]. The state-of-the-art XML structural indexes either need to be large to perform well or perform poorly as a consequence of saving space. Furthermore, the complexity of the XML data model leads to a much larger search space for XML query optimization [26] [128]. Finally, many types of XML structural indexes require a huge number of structural joins and match operations to establish a relation between two elements in XML data-trees.

1.1 Motivation

The rapid growth in XML databases has resulted in the need to efficiently query this XML data. One way to achieve fast retrieval of data is through indexing. XML structural indexes can be used to facilitate more efficient query processing and optimization [124].

There are many advantages to the XML data model compared with traditional data models like the relational model [16] [56]. The structure is integrated with the data in an XML document [5] [82], whereas, in the relational model the structure is defined in a separate relational schema. Therefore, it is easier to use XML as an intermediate language for exchanging data in the World Wide Web. Also, unlike the relational approach, the XML data model adapts easily to the evolution of the data structure in a database [127]. Finally, the XML data model is flexible for querying data. This kind of flexibility does not exist in SQL (Structured Query Language) [1].

Nevertheless, these advantages come with a cost. First, since the repetition of data is irregular due to missing and/or repeated arbitrary elements, its storage structure can be scattered over many different locations on the disk, which decreases the performance of XML queries [32]. Second, the flexibility of specifications of the XML queries (e.g. use of wild cards) adds to the challenge of indexing methods [130] [146]. Third, the fact that XML documents contain the data mixed with the structure poses a challenge to navigating the structural relationships among XML element sets [65].

One of the main differences between XML data and relational data models is the variety of structural relationships between various elements in XML data [26]. For example, Figure 1.1 contains a relational database schema and data (we mixed the data with the schema for simplicity). This database represents a small portion of an educational institute registration system.

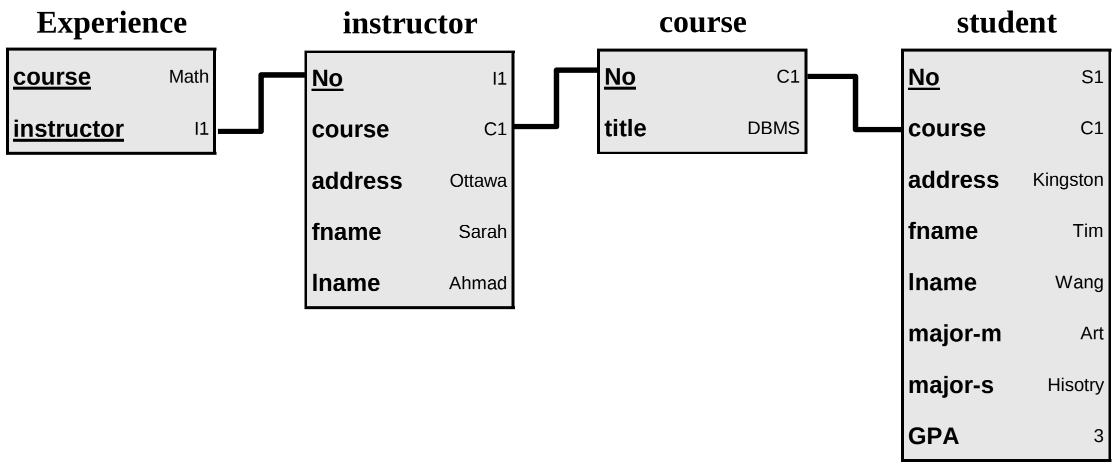


Figure 1.1 Simple relational database

Note that the complete data are well structured and can be linked through 3 primary/foreign key relationships. In this model we can access various sets of data through only 3 relationships. For example, which course is taken by which student?, what courses a specific instructor teach? … etc. This database can be represented as an XML database. One possible representation is illustrated in Figure 1.2.

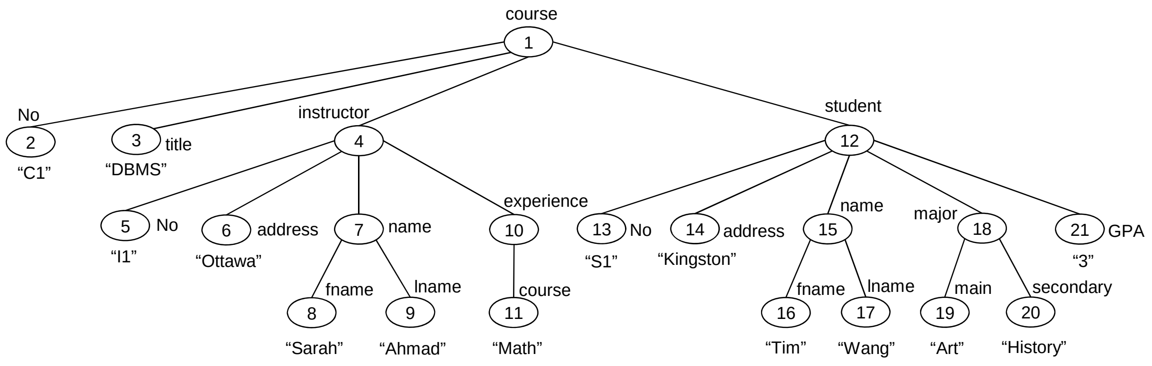


Figure 1.2 An XML data-tree for the relational database in Figure 1.1

Basically, the most used relationships between XML elements are ancestor, parent, sibling, child, and descendent relationships, which can be used to infer other types of relationships[1]. In our XML data model example in Figure 1.2 we have 14 leaf data elements. Each and every one of these elements can be associated with the other 13 elements through several relationships. So, there are at least 13x14 relationships that have to be indexed properly in order to manipulate and query this XML data efficiently. This adds more complexity to the XML data model. As a result, the creation of a *universal* structural index that reflects all of these relationships efficiently is a challenging task. In the relational approach, in contrast, the relationships are much more limited between different elements in different tables, and the data are well structured.

Labeling schemes can be divided into two categories: Interval labeling schemes and Prefix labeling schemes. Each type of scheme has advantages and disadvantages. Undesirably, each type of labeling' pros are the cons of the others. Research efforts to design a labeling scheme that works well in all environments have flourished lately.

1.1.1 Research Track

In an effort to address the problem of XML structural indexes, we first study the existing indexing techniques and analyse their strength and weakness. The best way to do that is to compare these indexing techniques by using common criteria that are applicable for all of them and can act as a benchmark. We therefore identified the following four comparison criteria:

[1] There are 13 relationships as identified by XPath – an XML query language that is recommended by the World Wide Web Consortium (W3C).

- *Retrieval power*, which includes the precision and completeness of the result, and the type of queries supported.
- *Processing complexity*, which covers topics related to the need to compute the relationship between elements (such as the parent-child and the ancestor-descendent relationships), the need for structural joins to answer a query, and the need for additional refinement steps to fine-tune answers.
- *Scalability* of the index and its *adaptability* to queries with different path lengths.
- *Update cost,* which is measured by the number of nodes that are touched during update.

Based on our findings, we came across many issues. These issues include the following. First, we identify structural joins as the bottleneck stage in XML query processing. Second, there is a trade-off between the size of an index and its answering power. Third, relational database management systems are promising media to store and retrieve XML data, but the current mapping approaches to map XML data into relational tables are still immature.

We believe that the labelling scheme used is the key factor in controlling the size of indexes and to reducing the number of joins required to evaluate queries. Some researchers combine different types of indexes to expedite query processing, such as combining node indexes with graph indexes. In this case, we believe that the integrated system does not have to have the structural information available in both indexes. The intuition behind relaxing the structure constraint in one of them – e.g. the node index - is to have more room for designing a labeling scheme that performs better in evaluating XML queries. Most of the present mapping approaches are based on nodes, edges, forward paths, and reverse paths. The last approach has proved to be the best, but often produces huge indexes. We believe that mapping XML data based on the path summary not only reduces the size of indexes by eliminating redundant path data, but also improves the query evaluation process by linking the indexed data nodes to their original structure.

XML data model is based on hierarchical structure, which is absolutely different from the relational data model. Therefore, we believe that if we want to use RDBMSs to store and retrieve XML data efficiently, we have to address two issues. First, the mapping scheme should reflect the original hierarchical structure. Second, an XML engine has to be used along with a relational engine to process queries. In this way, the XML engine optimizes queries according the structure of the data before shredding, and the relational engine handles the data stored in relational tables that pertain to queries under process.

1.2 Thesis Statement

We study and research the outstanding issues that are related to XML structural indexes. In the scope of this dissertation, and based on our findings in the areas of XML labeling schemes, native XML structural indexes, and XML-Relational structural indexes; we propose solutions to these outstanding issues. As part of the proposed solutions we implement: a labeling scheme called LLS; a native XML index structure called LTIX; and an XML-Relational index structure called UISX. We conduct our experiments to fine-tune our proposed systems in our laboratory. The experiments support our intuitions that guided our research. The experiments are also used for enhancing our findings.

1.3 Contributions

The thesis makes the following original research contributions:

- Develop a novel native XML index structure, which includes:
    - Designing a structural summary for XML data, and implementing an efficient construction and access algorithms to this structural summary.
    - Defining a framework to implement and use the native XML index structure.
    - Designing an efficient repository for the elements, attributes, and values of XML data that facilitates speedy retrieval.
- Develop a novel XML-Relational index structure, which includes:
    - Indexing the branching nodes to evaluate twig queries.
    - Designing and implementing an efficient relational schema to map shredded XML data using minimal storage space.
    - Developing a light-weight query processor to evaluate XML queries using native XML engine on top of SQL engine. The job of the native XML engine is to explore potential query optimization processes that are related to the structure of XML data, which can not be exploited by SQL engines. The SQL engine handles the XML-Relational data after shredding.

  - ▪ Designing algorithms to evaluate XML queries efficiently that do not use neither inequality comparison, nor "LIKE" operator, since SQL engine does not support them efficiently.
- Develop a novel suitable labeling scheme to be used with the above two index structures. This labeling scheme should perform well in comparison with existing labeling schemes.

1.4 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 presents the background on XML data models and the literature study on XML structural indexes. In Chapter 3, we present our LLS labeling scheme with a prototype to demonstrate the effectiveness of this labeling scheme. The LTIX native index structure is described in Chapter 4, which includes a discussion of various techniques for building the summary graph of the index structure, and the experimental validation of our indexing approach. The UISX XML-Relational index structure is illustrated in Chapter 5. It includes a discussion of the various mapping approaches. Finally, we summarize the contributions of our research with a critical assessment, discuss some of the future work directions, and conclude the thesis in Chapter 6.

1.5 Summary

The complexity of XML data model and flexibility of querying it creates many new challenges for the researchers in the area of XML structural indexes [127]. We address the problems of existing labeling schemes, which include the updating cost and the size of the labels. The dissertation presents a novel labeling scheme, which is designed to contain the most needed characteristic in order to work properly and efficiently. Also, the dissertation presents two novel indexing structures. The first is based on native XML structure and the second uses RDBMSs to store and query XML data. We start with laying out the motivation behind this research dissertation. Then we state our research hypotheses, the scope of our work, and the contributions of this research in the area of XML structural indexes. In the following chapters we provide the background study, detail presentations of XML structural indexes, LLS labeling scheme, LTIX and UISX index structures with experimental evaluations of their methodologies and prototypes, and finally summarize and conclude the thesis outlining some of the future work directions.

Chapter 2

Background and Literature Study

XML, which provides a flexible way to define semistructured data, is a de facto standard for data exchange in the World Wide Web. The trend towards storing data in its XML format has meant a rapid growth in XML databases and the need to query them. Indexing plays a key role in improving the execution of a query [124]. In this chapter we give a brief history of the creation and the development of the XML data model. We discuss the three main categories of indexes proposed in the literature to handle the XML semistructured data model and provide an evaluation of indexing schemes within these categories. Finally, we discuss limitations and open problems related to the major existing indexing schemes.

2.1 Background

XML documents can be represented as directed graphs [3], which consist of vertices and edges. For example, the directed graph in Figure 2.2 represents the XML document in Figure 2.1. The *mapping* of an XML document to a graph may result in an acyclic graph (e.g. Figure 2.2), which is tree shaped, or in a cyclic graph (if ID/IDREF tokens are used). While some indexes support all graph data [10] (cyclic and acyclic graphs), others only support tree-shaped data. In this section, we review four common models for semistructured documents and the XPath query language, which is used in this thesis to express queries.

2.1.1 Data Models

Gou and Chirkova [56] identify four basic data models to represent the hierarchical structure of XML documents: edge-labeled tree data model, node-labeled tree shaped data model, directed acyclic graph (DAG) data model, and directed graph with cycles data model.

2.1.1.1 Edge-Labeled Tree Data Model

Figure 2.2 is an example of an edge-labeled model for the XML document in Figure 2.1. Each edge represents an element or an attribute in the XML document. For example, *author* is an element, and *reviewer* is an attribute. The leaf nodes represent the values of the elements or attributes. For example, "Ahmad" and "Wang" are values for the *reviewer* attribute and *author* element, respectively. The same attribute name cannot be repeated under the same element. Attributes are unordered and cannot be nested as in elements. The element in the fifth line in Figure 2.1 is an example of an empty element.

```
<Bib>
    <book>
        <author>Tim</author>
    </book>
    <paper> </paper>
    <paper>
        <author>Sarah</author>
    </paper>
    <paper reviewer="Ahmad">
        <author>Wang</author>
    </paper>
</Bib>
```

Figure 2.1 XML document

Note that in a tree structure an element cannot have more than one parent. The same tag name can be repeated along a path (i.e. an element may have a child/descendent element and/or a parent/ancestor element with the same tag name(s)). This is known as recursion, which requires special attention during the evaluation process of an XML query.

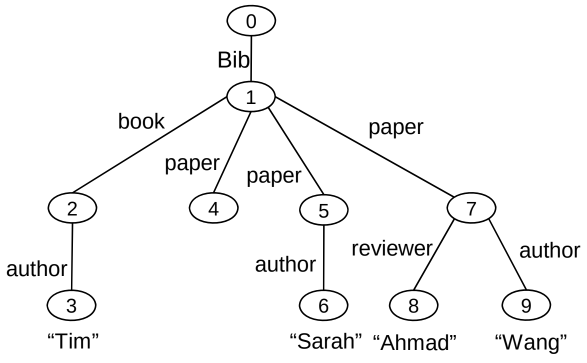


Figure 2.2 Edge-labeled data-tree

2.1.1.2 Node-Labeled Tree Data Model

Figure 2.3 is an example of a node-labeled data-tree for the XML document in Figure 2.1. As in the edge-labeled model, it contains three main components: elements, attributes, and values. The main difference is that a node in the node-labeled tree represents an element as opposed to an edge in the edge-labeled model. The hierarchal and nesting structure of both models is self-evident in the trees that they represent.

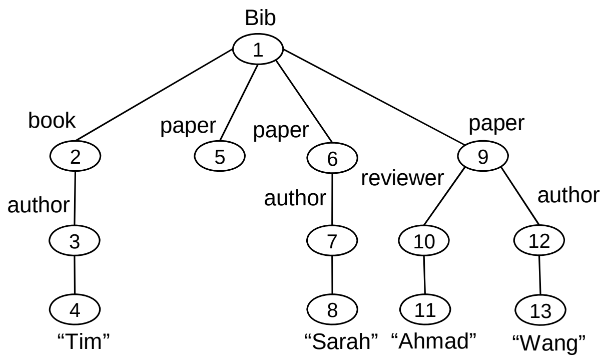


Figure 2.3 Node-labeled data-tree

2.1.1.3 Directed Acyclic Graph Data Model

Generally, the directed acyclic graph data model uses ID/IDREF tokens to identify an attribute type of an element. The ID/IDREF tokens are provided by the XML language via DTD. Figure 2.4 is a modified version of the XML document in Figure 2.1. Note the use of ID/IDREF and its effect on the corresponding DAG in Figure 2.5 (the dashed arrow from node 4 to node 2). Unlike the tree structure, a single node can be referred to by two or more elements in the DAG model (e.g. node number 2 in Figure 2.5). ID/IDREF is similar to the key/foreign key relationship in the relational data model.

```
<Bib>
    <book ID=1>
        <author>Tim</author>
    </book>
    <paper reference=1> </paper>
    <paper>
        <author>Sarah</author>
    </paper>
    <paper reviewer="Ahmad">
        <author>Wang</author>
    </paper>
</Bib>
```

Figure 2.4 XML document with ID/IDREF

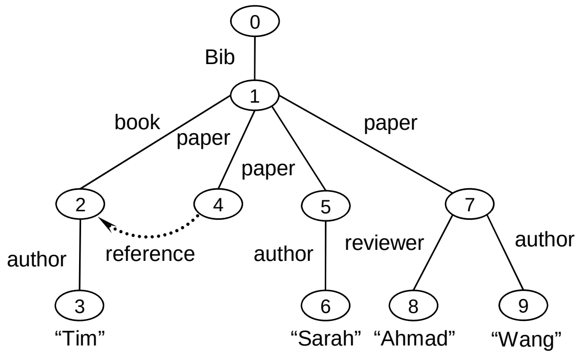


Figure 2.5 Directed acyclic graph data model

2.1.1.4 Directed Graph with Cycles Data Model

If we add an IDREF from the *book* element ("recommend=2", line number 2 in Figure 2.6) to the *paper* element ("ID=2", line number 5), a cycle is formed. This is also popular in XML, but it adds more complexity in query processing of XML data. The result is a directed cyclic graph as illustrated in Figure 2.7.

```
<Bib>
    <book ID=1 recommend=2>
        <author>Tim</author>
    </book>
    <paper ID=2 reference=1> </paper>
    <paper>
        <author>Sarah</author>
    </paper>
    <paper reviewer="Ahmad">
        <author>Wang</author>
    </paper>
</Bib>
```

Figure 2.6 XML document with ID/IDREF

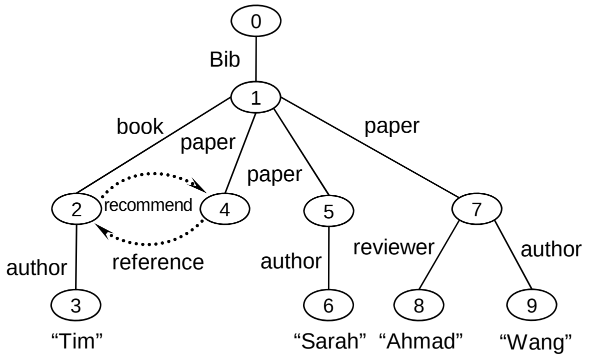


Figure 2.7 Directed graph with cycles data model

2.1.2 X-Path

Many APIs (Application Program Interfaces) have been proposed to access XML data, such as the standard Document Object Model (DOM) [53] [83] and Simple API for XML (SAX) [85]. DOM has been defined to enable XML to be manipulated by software [53]. The DOM defines how to translate an XML document into data structures and thus can serve as a starting point for any XML data model. DOM and SAX are language-independent programmatic APIs [50]. Whereas DOM creates an in-memory representation of an XML document, SAX provides stream-based access to documents. As a document is parsed, events are fired for each open and close tag encountered. Thus, in contrast to DOM, SAX only supports read-once processing of documents.

Nevertheless, neither one of these APIs provides enough capabilities to manipulate and query XML data. Motivated by this fact, query languages such as XPath (XML Path Language) [33] and XQuery [16] were proposed. XPath supports thirteen types of relationships or axes including child ("/"), descendant ("//"), parent, ancestor, ancestor-or-self, descendant-or-self, following, following-sibling, preceding, preceding-sibling, attribute, self, and namespace. In this thesis we concentrate on the child "/ " and descendent "// " axes. Furthermore, our proposed approaches in this thesis are capable of supporting these two axes. Both XQuery and XPath were developed and recommended by the W3C. Furthermore, a version of XQuery (Version 1.0, 1997) is based on XPath [16].

XPath provides operators for path traversals in an XML tree-shaped document. Path traversals result in a collection of subtrees (forests), which may be repeatedly traversed until a designated destination node is reached. Starting from a specific node, an XPath query navigates its input document using a number of location steps. For each step, an axis describes which document nodes (and the subtrees below these nodes) form the intermediate result forest for this step using one of the above mentioned 13 axes.

An XML query may be either a simple single path query with or without a recursion ("// " descendent axis), or a multiple path (twig) query with or without a recursion (please note that single path query is also called path query, for short, in this dissertation). Furthermore, an XML query may have a zero, one, or multiple predicates. A twig query specifies patterns of selection on multiple elements related to one another by a tree structure. Next, we review a few examples of an XPath XML queries. Query 2.1 below is an example of a single path query.

Query 2.1: */Bib/paper/author*

If we run this query against the XML document in Figure 2.3, it returns the results {"Sarah", "Wang"}, which are the values of the *author* elements under the *paper* elements under the *Bib* element. Query 2.2 is an example of a recursive query, which illustrates the use of the descendent axis.

Query 2.2: */Bib//author*

This query returns a set of values {"Tim","Sarah","Wang"}, which represents all *author* elements under the top-level *Bib* element. Query 2.3 is an example of a twig query that has a predicate.

Query 2.3: *//paper[/reviewer="Ahmad"]/author*

This query asks for the *author* of a *paper* that has a *reviewer* named "Ahmad," and the query returns the author "Wang." This query demonstrates the flexibility that XPath provides, which is not available with the relational data model. It allows us to query about a paper without concern for where the paper is located within the tree structure. However, it adds more complexity to the query language where an effort has to be made to locate the *paper* element through some indexing scheme, or else an exhaustive search has to take place if an index is not available.

In query 2.3 we also see an example of using a predicate in an XPath query. Multiple predicates could be used in an XPath query. Path patterns for the above three XPath queries are shown in Figure 2.8. In this Figure, an oval represents an element, the edges between elements represent the parent-child relations, the edges that are marked with an "=" sign represent the ancestor-descendent relationships, and the nodes with the question marks are the output nodes [56].

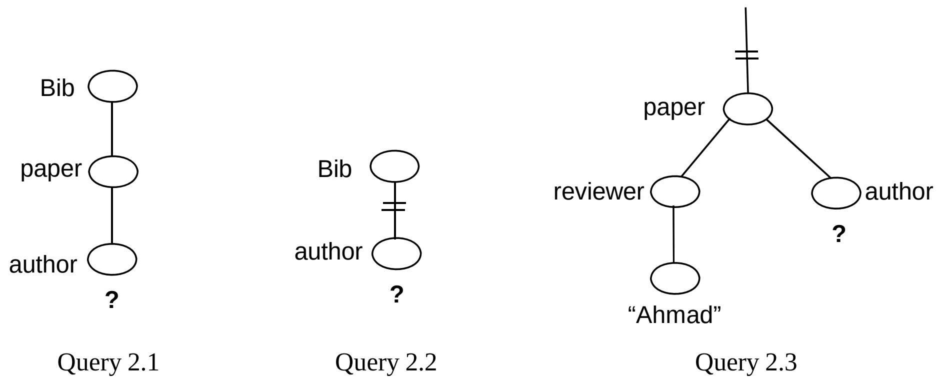


Figure 2.8 Schematic representation of XPath queries.

2.2 Structural Indexing Schemes for XML Data

Some XML databases structural indexes, such as the graph indexes, are analogous to the schema of a relational database. Both of them reflect the relationship between different parts of the data, and they are used to validate the legitimacy of a query before executing it. For example, XML graph indexes are used to determine if an XML path exists, before going any further in the query processing for both single and twig path queries [108] [123]. In this section, structural indexes for XML data are analyzed in detail.

Generally, structural indexes can be grouped into three categories:

- *Node indexes* [57] [78] [143] depend on labeling schemes [31] [60] including interval labeling [40] and prefix labeling [81] [100] [119] [138].
- *Graph indexes* [62] include indexes that cover either single path queries only [32] [37] [54] or both single and twig path queries [67]. We divide graph indexes in this thesis into three types depending on their deterministic property and bisimilarity direction(s) (see Section 2.2.3: Graph Indexing Schemes).
- *Sequence indexes* [104] [129] [130] interpret queries as structure-encoded sequences and search for a match in the structure-encoded sequences of an XML document.

Please note that the term *path index* is used in the literature to refer to different things. Sometimes it refers to graph indexes in general or to specific types of graph indexes (the deterministic graph indexes and the non-deterministic backward bisimilar indexes), and sometimes it may refer to some types of node indexes (prefix indexes). In this dissertation we prefer not to use the term *path indexes* and use the specific terms above in order to eliminate any ambiguity.

2.2.1 Criteria for Evaluation of Structural Indexing Schemes

We evaluate the indexing schemes according to a common set of criteria. These criteria are chosen in a way to help users decide which indexes are most suitable for their needs by identifying the characteristics that these indexes support, such as accuracy, completeness, response time, scalability, and adaptability. We use the following criteria:

- *Precision:* When a query is evaluated, the results returned may be complete and precise, or they may require further processing. Obviously, the first option is more efficient if the measurements of time taken to produce the initial answer for the two options are approximately equal. A structural index is precise if and only if the returned answer does not contain any incorrect answers.
- *Recall:* This is the probability that all relevant documents are retrieved by the query. If the recall achieved is 100%, we say that the result is complete.
- *Processing complexity:* This criterion covers different kinds of complexity depending on the type of indexing scheme that is used. It covers the primary processing procedure as well as additional join processing. Complexity criteria for each indexing scheme will be discussed individually.
- *Scalability*: Large indexes may involve many Input/Output (I/O) accesses. These accesses increase the processing time of a query. Some indexes expand linearly with the size of the source data, while others increase exponentially with the size of the data. The second type imposes restrictions on the data growth.
- *Adaptability:* Graphical indexes partition the data into equivalence classes based on their determinism and bisimilarity (backward bisimilarity, or forward and backward bisimilarity). Two nodes are backward bisimilar if they share the same incoming paths and forward bisimilar if they share the same outgoing paths. The bisimilarity can be specified by a factor k. Two nodes are backward k-bisimilar if they share the same incoming paths of a length = k. Setting the value of k to a small value results in a small index,

while a large value of *k* results in a large index. The length of the path in queries varies depending on the users' needs. If a graph index is used regularly to evaluate short-path queries, then a small *k*-value index is sufficient. In contrast, long-path queries need a large *k*-value index. Based on these observations, and depending on the queries, it would be useful if the size of the index could be adjusted by a given parameter *k* that represents the length of bisimilarity according to the users' need.

- *Type of queries supported:* The two types of XML queries in general are single path and twig path queries.
- *Update cost of insertion of a node or a subtree*: The nodes in a given tree index have to be maintained in a certain organization in order to reflect ancestor-descendent, parent-child, and sibling relationships. When a new node is inserted into the tree, these relationships have to be preserved. Consequently, the index has to reflect its position with regard to these relationships, which adds more complexity, especially if there are no gaps in the scheme that is used to label nodes. We study two types of updates [139]: (1) the insertion of a node, which represents a small incremental change for an edge addition (for all indexing schemes); (2) the insertion of a subtree, which represents the addition of a new file (for some indexing schemes).

2.2.2 Node Indexing Schemes

Node indexes hold values that reflect the nodes' positions within the structure of an XML tree. They can be used to find a given node's parent, child, sibling, ancestor, and descendent nodes. These values can be used to evaluate single path and twig path queries. Paths are evaluated through many steps. At each step, a structural join is performed between two nodes starting from one end of the path and finishing at the other end [6] [78] [143].

Labeling (numbering) schemes were used prior to the creation of XML to reproduce the structure of a tree [40]. Two of the most widely used labeling schemes are interval (a.k.a. region) labeling [8] [29] [72] [78] [116] [135] [143] and prefix (a.k.a. path) labeling [48] [60] [66] [80] [99] [119]. In the following, we take the (*Beg, End*) labeling scheme as an example of the interval labeling and the *Dewey* code scheme as an example of the prefix labeling.

2.2.2.1 Criteria for Evaluation of Node Indexes

In addition to the general evaluation criteria listed above, we refine the processing complexity criterion into the following specific criteria.

Processing complexity:

- *Relationship computation:* To confirm a relationship between two given nodes, certain operations have to be performed. These operations depend on the type of the relationship. They also depend on the type of the labeling scheme that is used.
- *Relationships supported:* Basically there are three types of relationships:
  - *Ancestor-descendent relationship*: This relationship is needed to evaluate queries with the "//" axis.
  - *Parent-child relationship*: It is useful to evaluate queries with the "/" axis.
  - *Sibling relationship*: In some cases, a group of sibling nodes form an answer for a twig query.

- *Ability to infer parent/ancestor and child/descendent nodes*: There are two approaches for solving queries, especially the ones with predicates, that is, top-down and bottom-up. A bottom-up approach is useful when the parent/ancestor nodes of a matched leaf node, for a given query, can be inferred from the matched leaf node. Also, identifying child/descendent nodes is helpful when the top-down approach is used to evaluate a query.
- *Data type used in indexing scheme*: Comparing different data types involves different algorithms with different operations. As an illustration, comparing two numbers usually requires less time than that of comparing two sequences of strings.

2.2.2.2 Interval Labeling Scheme

The (*Beg,End*) labeling scheme is an example of interval labeling. Zhang et al. [143] introduce it to index the elements in a document. It assigns a pair of numbers to each node in an XML document according to its sequential traversal order as follows. Starting from the root element, each element, attribute of an element, value of an attribute, and value of an element is given a *Beg* number according to its sequential position in the document. When we reach the end of a tag, an attribute, or an attribute value, we assign to that tag, attribute, or attribute value an *End* number (which is equal to the next available sequential number) before moving to a new element in the XML document. When the value of an element is a leaf node, the *Beg* number of this value is equal to the *End* number. Figure 2.9 is an example of (*Beg,End*) labeling scheme for the XML document in Figure 2.1. The beginning and the ending numbers imply the positions of the opening tag (<..>) and the closing tag (</..>), respectively, in an XML document.

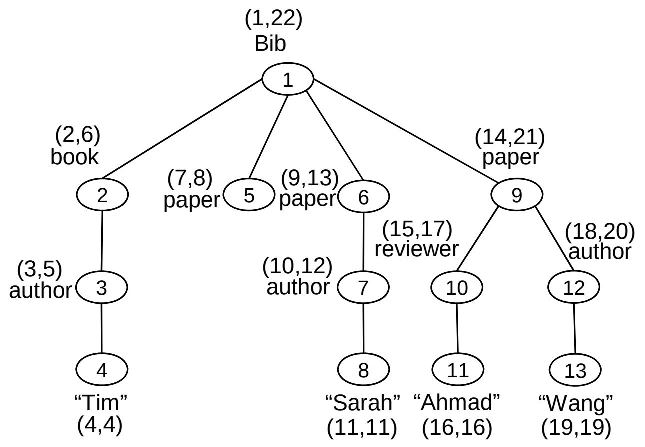


Figure 2.9 (*Beg,End*) labeling scheme

This labeling scheme enables us to find the ancestor-descendant relationship as indicated in *property 1* below. A *Level* is added to the (*Beg,End*) label to form a node-triplet identification label (*Beg,End,Level*) for each node in the tree, where *Level* represents the depth of an element in the tree [143]. This triplet identification label is used to infer the parent-child relationship as indicated in *property 2*.

Property 1 - Ancestor-descendant relationship: In a given data-tree, node x is an ancestor of node y if $x.Beg < y.Beg < x.End$. For example, in Figure 2.9 node (1,22) is an ancestor of the node (3,5).

Property 2 - Parent-child relationship: In a given data-tree, node x is a parent of node y if $x.Beg < y.Beg < x.End$ and $y.Level = x.Level + 1$. For example, in Figure 2.9, node (1,22,1) is a parent to the node (2,6,2).

The (*Beg,End*) scheme can be used to evaluate a twig query by using structural joins [113]. The relations that are supported by the node approach are mainly the parent-child "/ " and the ancestor-descendent "//" relationships. The (*Beg,End*) labeling scheme is used to infer the relationship between two nodes at a time. It requires two comparisons to infer any of these two relations. The number of joins required to evaluate an XML query using a node index is equal to the number of nodes in the query minus one, which is high for large twig queries.

The (*Beg,End*) labeling scheme can be used to evaluate both single path queries and twig path queries [113]. For a given query, the relationship between any two nodes within a path in the query is investigated separately because this indexing scheme's granularity is defined at the level of each node and hence the answer for a given query will be precise and complete.

Since the nodes' index numbers are chosen sequentially, or randomly in an increasing order, and the tree is not necessarily balanced, there is no way to locate the siblings of a given node, using only the knowledge of its index numbers. Furthermore, the exact ancestor and descendent index numbers of a node cannot be inferred. It is possible to know the range within which the parent/ancestor or the child/descendent nodes are located, but the exact number of these nodes cannot be determined.

Temporal XML databases [86] [35] are based on persistent (immutable) labeling schemes. Once a node is given an index number (e.g. "*Beg,End*" numbers), it remains unchanged throughout its lifetime. Persistent labeling is useful for examining changes to the contents of a source data over time by reviewing historical data. The paper by Cohen et al. [34] is an example of the early work in this area.

Unlike a prefix labeling scheme, which we explain in the next section, the interval labeling scheme is best used for immutable encoding. Some *durable* schemes, for example Li and Moon [78], suggest leaving gaps between the interval values for new nodes to be inserted. After filling these gaps, renumbering or other solutions are required. Cohen et al. [34] proved that persistent labeling, which preserves the order of an XML tree, requires *O(n)* bits per label where *n* is the size of the tree. The complexity is measured in the size of the interval labels because this size determines the total size of the index. It is desirable to keep the used number of bits small enough so that the index can fit in memory. Several researchers including Silberstein et al. [116] and Chen et al. [29] have designed dynamic labeling structures for interval indexes that allow relabeling by using only *O(log n)* bits per label.

Interval labeling schemes require modest storage space. Regardless of the depth of the data-tree, each node is represented by only two numbers, and we can determine the relationship between any two nodes in fixed time by using comparison operations between the index numbers. Nevertheless, updating the labeling scheme of these types of indexes is costly. When a new node is inserted into the tree, then all the nodes in the tree, except the left sibling subtrees of the inserted node, have to be updated.

Surveying all the variations of interval labeling is beyond the scope of this chapter. In the following, we list a few of the variations. Dietz [40] pioneered the labeling of an ordered tree [56] [78]. He used (*Pre-order, Post-order*) numbers to label the nodes of a data-tree. Pre-order sequence is based on traversing the tree recursively from the root *R* to subtrees rooted at *R* in a depth-first direction. Post-order sequence is based on traversing the tree in an opposite direction to that given in pre-order sequence. A vertex *x* is an ancestor of *y* if and only if *x* occurs before *y* in the pre-order traversal of the tree and after *y* in the post-order traversal. Li and Moon [78] propose the (*Order,Size*) labeling scheme. The *Order* part is based on a pre-order traversal, and the *Size* part is an estimate of the number of the child/descendent nodes for a given node. This labeling scheme leaves room for expansion in order to avoid relabeling of the data-tree in case of insertion. Relabeling may be delayed, but eventually it is required. It occurs more often if the data distribution in the tree is skewed.

Tatarinov et al. [119] discuss the possibility of using *real numbers* (rational numbers) instead of integers to represent a position in their proposed global order of XML trees and discarded this idea because there is a finite number of values between any two real values stored in the computer and using real values instead of integers does not make much difference. Later, Amagasa, Yoshikawa, and Uemrua [8] used rational numbers instead of integers to represent a region (interval) in node indexing. Similar to the (*Order,Size*) labeling scheme [78], the rational number approach only avoids node relabeling as much as possible. If the number of insertions exceeds a specific limit, the nodes have to be relabeled. Wu et al. [135] propose a novel labeling scheme that uses *prime numbers* to label nodes in an XML tree. In this approach, each node label can only be divided exactly (without remainder) by its own ancestor(s).

2.2.2.3 Prefix Labeling Scheme

Dewey code labeling (Dewey labeling for short), which is an example of a prefix labeling scheme, is another labeling scheme that was originally made for general knowledge classification [100]. Tatarinov et al. [119] first used it for XML tree-shaped data.

Each node is associated with a vector of numbers that represents the node-ID path from the root to the designated node. In addition to being classified here as a node index type, it can also be considered as a path index since each node is represented as a complete path from the root to the indexed node.

Figure 2.10 is an example of the Dewey labeling scheme for the XML document in Figure 2.1. Each node label represents the node location within a path by including its ancestors' coding as a prefix (vertical coordinate), and it also includes the node number within the siblings of the same parent (horizontal coordinate). The level is implicitly included by counting the number of segments that are separated by a delimiter (dot in our example in Figure 2.10) in the Dewey labels.

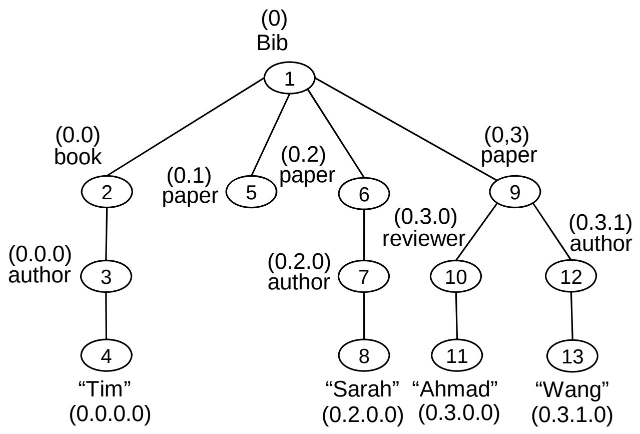


Figure 2.10 Dewey labeling scheme

To decide if a parent-child or an ancestor-descendent relationship exists, we perform a prefix matching operation on the index string. In a given data-tree, node *x* is an ancestor of node *y* if the label of node *x* is a substring of the label of node *y*. For example, node (0.3) is an ancestor of node (0.3.1.0). Unlike the (*Beg,End*) labeling scheme, the Dewey labeling scheme does not require any additional information in order to evaluate the parent-child relationship. For example, it is easy to see that node (0.3) is the parent of node (0.3.1).

The sibling relationship can be computed in the same way without the need for any additional information (e.g. level number or parent ID). The Dewey labels provide direct support for the sibling relationship. In a given tree, node x and node y are siblings if nodes x and y have the same number of fragments in their labels (call it n) and x.prefix = y.prefix (where the prefix length is equal to n minus one). For example, node (0.3.0) and node (0.3.1) are siblings.

Dewey labels are much easier to update than (*Beg,End*) labels. When a new node is inserted, only the nodes in the subtree rooted at the following sibling need to be updated [119]. However, its storage size increases with the depth of the tree. Furthermore, as the depth increases, it becomes more costly to infer the parent-child or the ancestor-descendent relationship between any two arbitrary nodes because the string prefix matching becomes longer.

Fisher et al. [48] propose a dynamic labeling approach that can be applied to Dewey labels with identifiers of size $O(log\ n)$ when there is type information in the form of a DTD or Schema, where n is the size of the database. Similar to all labeling schemes, immutable Dewey labeling requires $O(n)$ bits per label [34].

It is easy to infer the exact ancestors or descendents of a given node in Dewey labeling scheme indexes. For example, in Figure 2.10 the ancestors of the node (0.3.1) are the nodes that start with a (0.3) or (0) prefix, and the descendents are the nodes that start with the (0.3.1) prefix, such as node (0.3.1.0). Since the complete path is recorded within a node index, Dewey labeling scheme indexes return a precise and a complete answer for both path queries and twig queries. Path and twig queries need join operations in order to be evaluated, specifically the number of nodes in the query minus one join operations are required.

Many variants of prefix labels are proposed in the literature. O’Neil et al. [99] propose the ORDPATH labeling scheme that is similar to the Dewey labeling scheme, except that the child nodes of a given parent node are labeled by using odd numbers, and even numbers are used later for new insertion. In the GRoup base Prefix (GRP) labeling scheme [80] the labels consist of two parts, namely, group ID and group prefix. Doung and Zhang [42] propose Labeling Scheme for Dynamic XML data (LSDX), where the labels are a combination of numbers and letters. LSDX support the ancestor-descendent relationship as well as the sibling relationship between nodes. GRP and LSDX labeling schemes are persistent, therefore their label sizes can reach *O(n)* bits per label in the worst case, where *n* is the number of nodes in the tree.

2.2.2.4 Summary of Node Indexes

Table 2.1 contains a summary of the two types of labeling schemes that are used to form node indexes. The precision of an index scheme is either precise (does not return any false answers) or imprecise (may contain some false answers along with the correct answers). If the recall achieved is 100% then the result is complete, otherwise it is incomplete. Relationship computation is fixed if we can determine the relationship between any two arbitrary nodes in a fixed time, which may depends on the depth of the data-tree. The relationships supported are ancestor-descendent, parent-child, and sibling relationships. The data type is either a number or a string. The types of queries supported by these node indexing schemes are path and twig queries. The evaluation of both types of queries usually require join operations. The maintenance cost of the indexes depends on the number of elements and whether or not the index is mutable or immutable.

Table 2.1 Comparison of interval labeling scheme with prefix labeling scheme

No.	Criteria		Interval Labeling (Beg,End)	Prefix Labeling (Dewey)
1	**Precision**		Precise	Precise
2	**Recall**		Complete	Complete
3	**Computation Complexity**	**Relationship computation**	Fixed	Directly proportional to depth increase
		Relationship supported	- Ancestor/Descendent - Child/Parent (if "Level" is available)	All
		Can infer exact ancestor & descendent nodes	No	Yes
		Data type	Numerical	String
4	**Size/Scalability for increasing depth**		Linear	Exponential
5	**Type of queries supported <u>efficiently</u> (without joins)**		None	None
6	**Maintenance cost**	**Mutable**	O (log n)	O (log n)
		Immutable	O (n)	O (n)

Both types are equivalent with respect to precision, completeness (recall), and maintainability. However, they differ with respect to the other characteristics. The (*Beg,End*) labeling scheme requires fixed time to compute a relationship between any two arbitrary nodes for two reasons. First, it uses integer values to index the nodes. Second, the size of the label that is used to index each node is fixed depending on the depth of the tree. On the other hand, in Dewey labeling schemes, the time that is required to compute the relationship between any two arbitrary nodes is directly proportional to the depth of the nodes for two reasons. First, Dewey labeling schemes use strings to represent labels instead of integers. Second, the labels' size increases as the depth increases. Unlike (*Beg,End*) labels, each Dewey label contain the root path (the path from the root to the designated node) information. Therefore, with Dewey labels, we can infer any node's parent-child or ancestor-descendent from the label of the node. Finally, prefix labels are often easier to update than interval labels, although, the cost of maintaining prefix labels can be the same as the cost of maintaining interval labels in the worst case.

2.2.3 Graph Indexing Schemes

A graph index (a.k.a. summary index) is a structural path summary [36] that can be used to improve query efficiency, especially for single path queries. It is also capable of solving twig queries but with an additional cost of multiple join operations.

Graph indexes consider paths, during query evaluation, as a whole path instead of dealing with each node in the path separately. A subsequent step is needed to join single paths together in order to evaluate a twig query. In contrast to node indexes, the number of joins is reduced during query processing and consequently query performance is improved.

Graph indexes have been categorized according to many criteria. For example, Gou and Chirkova [56] group them into two classes, path indexes, which are able to cover single path queries (such as strong DataGuides and 1-index), and twig indexes, which are able to cover twig queries (such as F&B-index). Graph indexes can also be categorized according to their path exactness [103]. Some schemes are exact such as strong DataGuide, Index Fabric, 1-index, and F&B-index; while others are approximate such as approximate DataGuide, A(k)-index, D(k)-index, and $(F+B)^k$-index.

Our classification considers the following important properties of an index:

- *Path determinism*: If the index tree is a Deterministic Finite Automaton, then the paths of the tree are considered to be deterministic paths. Given a particular input (tag name) to a particular node in a tree, it will always produce the same single output (path) if the system is deterministic, and it may produce several similar outputs (paths) if the system is non-deterministic. This feature assures that every distinct path in an index graph is represented only once. Otherwise, multiple identical paths may exist in the index, which may add to the complexity of query evaluation. Deterministic indexes guarantee uniqueness of paths, and non-deterministic indexes guarantee the uniqueness of elements.
- *Bisimilarity*: There are two types of bisimilarity, namely, forward and backward bisimilarity. Two nodes are backward bisimilar if they share the same incoming paths. Two nodes are forward bisimilar if they share the same outgoing paths. The direction of bisimilarity significantly affects the size of an index and the answering power of an index to a given query. Non-deterministic graph indexes with only backward bisimilarity tend to have lower accuracy (which is corrected by some post-processing steps) but their sizes are minimal. In contrast, graph indexes with forward and backward bisimilarity have higher accuracy and cover twig queries, but their sizes are larger than those of backward bisimilar indexes.

Based on path determinism and bisimilarity, we classify graph indexes into the following categories:

- *Deterministic graph indexes:* This includes strong DataGuides [54], approximate DataGuide [55], and Index Fabrics [37].
- *Non-deterministic graph indexes with backward bisimilarity:* This includes 1-index [90], A(k)-index [70], and D(k)-index [28].

- *Non-deterministic graph indexes with forward and backward bisimilarity:* This includes F&B-index [56] [2], $(F+B)^k$-index [67], disk-based F&B-index [131], and AB-Index [145].

Gou and Chirkova's [56] classification combines our first two groups into one that covers single path queries. Their classification for graph indexes is based on the type of queries (path or twig) an index covers, while our classification of XML graph indexes is based on their deterministic property, in addition to forward and backward bisimilarity.

Deterministic indexes guarantee uniqueness of paths, and non-deterministic indexes guarantee the uniqueness of elements. Therefore, deterministic indexes are suitable for single path queries (where the complete path is known). For example, to evaluate the query "*/P/A*" over the deterministic strong DataGuide index in Figure 2.11(B) we have to traverse one path only. In contrast, non-deterministic graph indexes may lead to traversing more than one index path to evaluate a single path query. For example, to evaluate the same query as described above over the non-deterministic 1-index in Figure 2.11 (C) we have to traverse more than one path that satisfies the query.

Non-deterministic graph indexes, on the other hand, represent every value in the source data only once in the index tree, while deterministic graph indexes may have the same value in the source data repeated in more than one location in the index tree. For example, node "9" in the deterministic strong DataGuide index in Figure 2.11 (B) is listed twice, while the non-deterministic 1-index in Figure 2.11 (C) has it listed only once. Furthermore, deterministic indexes may grow exponentially in the size of the original data (due to repetition of nodes), while non-deterministic indexes grow linearly [90]. Based on this discussion, in addition to fact that the term *path indexes* are used ambiguously in the literature to refer to absolutely different types of indexes, we use determinism as one criterion to classify graph indexes.

The other criterion that we use to classify graph indexes is the direction of bisimilarity. This criterion further subdivides the non-deterministic indexes into backward, and forward and backward bisimilar indexes. The direction of bisimilarity significantly affects the size of an index and the answering power of an index for a given query. Non-deterministic graph indexes with only backward bisimilarity tend to have lower accuracy (which is corrected by some post processing steps) but their sizes are minimal. In contrast, graph indexes with forward and backward bisimilarity have higher accuracy and cover twig queries, but their sizes are larger than those of backward bisimilar indexes.

We elaborate the development of graph index schemes according to these three classes and analyze the schemes using the general criteria given earlier. Please note that all graph indexing schemes provide a complete answer for both single path queries and twig path queries. They do not require extra joins to evaluate the single path queries but they require join operations to evaluate the twig queries (except F&B indexes).

2.2.3.1 Deterministic Graph Indexes

In deterministic graph indexes, each unique path in a data graph is listed once in the summary graph, and every path in a summary graph has at least one matching path in the data graph. Three indexing schemes of this type are strong DataGuides, approximate DataGuide, and Index Fabrics.

DataGuide

Goldman and Widom [54] present one of the early structure summaries called a strong DataGuide. In this scheme, the nodes in the source data are partitioned based on their root path, that is, the path from the root to the indexed node. The graph index (a.k.a. structure summary) of an XML data-graph is a strong DataGuide if it fulfills two conditions:

- Every distinct root path in the source data appears only once in the graph index.
- All the paths in the graph index have at least one matching root path in the original source data. In other words, there are no invalid paths in the graph index.

Figure 2.11 contains an XML data-tree and its associated graph indexes. To simplify the comparison between different schemes in Figure 2.11, we assume an edge-labeled graph structure, use numbers inside the nodes to represent the node IDs, and use letters to represent the elements (tag types) of the source XML data. The letters (*B*,*P*,*A*, and *R*) in Figure 2.11(B-F) stand for *book*, *paper*, *author*, and *reviewer* in Figure 2.11(A), respectively. Figure 2.11(A) is a modified version of the XML data-tree in Figure 2.2. The difference is that two edges are inserted (represented by the dashed lines in Figure 2.11(A)). The first edge connects nodes "4" and "3", and the second edge connects nodes "5" and "9". These edges transform the tree-shaped data in Figure 2.2 into directed acyclic graph-shaped data. Unlike node indexes, graph indexes are capable of supporting DAG data such as in Figure 2.11(A).

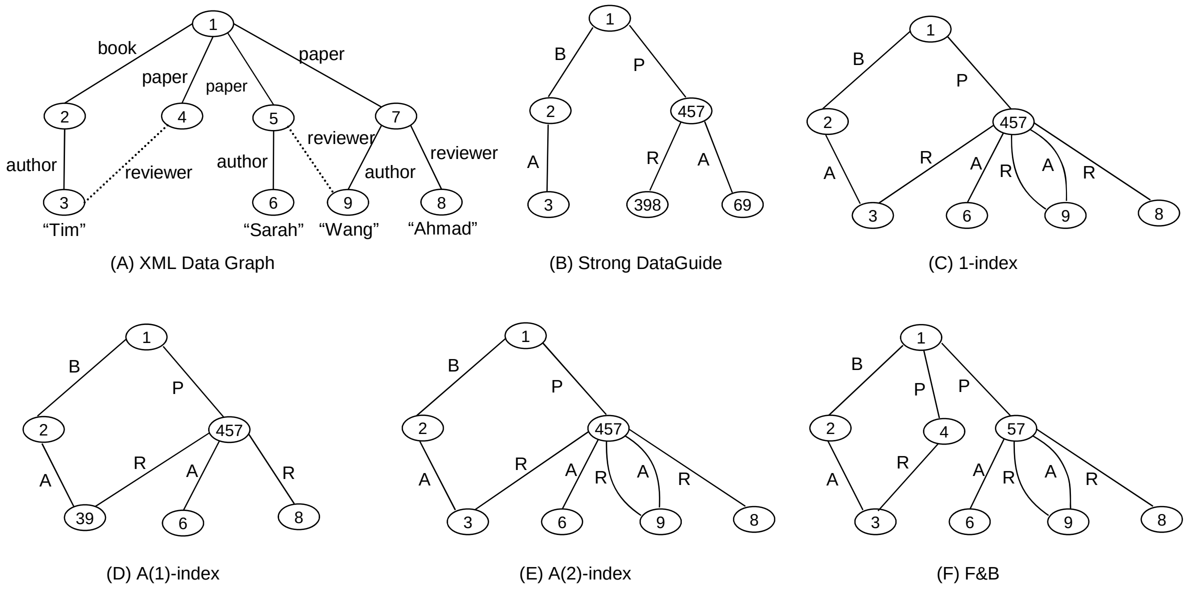


Figure 2.11 XML data-tree and its corresponding graph indexes

The graph index in Figure 2.11(B) is a strong DataGuide for the data in Figure 2.11(A). Note that node number "3" occurs in both the "*/B/A*" and "*/P/R*" paths. Node number "9" occurs in both the "*/P/R*" and "*/P/A*" paths. One may argue that being deterministic is an advantage of the strong DataGuide structure index. Nevertheless, a node's repetition is directly proportional to the existence of multiple parent nodes and cycles in the source data. In the worst case, the structural index size may exceed the original size of the data and hence it may lose its essential characteristic of a summary. In the case of DAG data, the size may be exponential in the size of the original data. Tree-shaped XML data, on the other hand, requires storage space, in the worst case, equal to the size of the data itself.

Strong DataGuides are capable of giving a complete and precise result for single parent-child path queries [67] such as *"/B/A"* in our example, which returns the node {3}. They are also complete and precise for ancestor-descendent path queries. For instance, the query *"//R,"* in our example, returns the nodes {3,8,9}.

Strong DataGuides are complete for twig queries but not precise [67]. For example, evaluating query *"/P[/A]/R,"* which returns an *R* node that has a *P* parent node and an *A* sibling node, over the strong DataGuide index in Figure 2.11(B) returns index nodes {3,8,9}. This answer is complete because the returned set includes the correct answer {8,9}, but it is not precise as node {3} does not belong to the correct answer.

The complexity of maintaining strong DataGuides depends on the structural effect of the updates. Updating strong DataGuides could be as simple as inserting a new leaf into tree-structured data, which requires only one target set to be recomputed and one new object to be added to the strong DataGuide. In the worst case, updating a tree with a subgraph of structured data that has loops and sharing may incur recomputation of a large portion of the strong DataGuide. An edge insertion update requires touching a number of nodes and edges that is equal to $O(n + m)$ in the worst case, where n is the number of nodes (objects) and m is the number of edges of a strong DataGuide. Please note that from this point forward DataGuide alone stands for strong DataGuide.

Approximate DataGuide

Experiments have shown, in general, that the size of the strong DataGuide is much smaller than the original database. There are cases, however, where the size of the strong DataGuide is unreasonably large (e.g., for cyclic data). Approximate DataGuide (ADG), which is proposed by Goldman and Widom [55], minimizes the size of strong DataGuides. ADG ignores the second requirement of the strong DataGuide, but maintains the first one. Therefore, it ensures that every distinct root path in the data source appears exactly once in the ADG, but it does not ensure that all ADG paths exist in the original data. Hence, an ADG may have false positives but never false negatives, so that all correct paths are guaranteed to exist in addition to some false paths. Experiments demonstrate that there is a trade-off between the size of ADG and its accuracy. In general, strong DataGuide characteristics are applicable for ADG, except that the size of the ADG is often smaller.

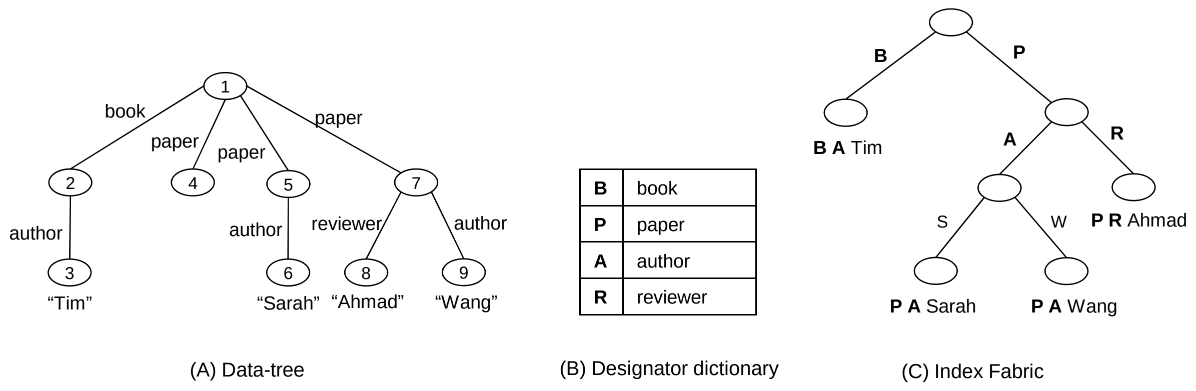
Index Fabric

Index Fabric is proposed by Cooper et al. [37] as a solution for very large indexes that may not fit in memory. Index Fabric utilizes its paging capabilities to solve the size problem. It uses prefix-encoding to represent paths as strings. These strings are classified and sorted by a special index called the Index Fabric. The index structure is designed specifically for complete path queries that start from the document root node. Other paths such as descendent path queries "//" require a post-processing stage and expensive index lookups. The notion of refined paths (template paths) is proposed by the authors to solve this problem. However, the refined paths are not dynamic and need to be determined prior to index creation and loading time.

The Index Fabric indexes both paths and values in a tree. As an example, each edge of the data-tree in Figure 2.12(A) (which is the same as the XML data-tree in Figure 2.2) is given a designator as illustrated in Figure 2.12(B). The edge labels along with the content of the data-tree are combined at the leaf nodes to form a graph index for each value in the tree. Note that compression is used to minimize the size of the tree. For example, in Figure 2.12(C), since *book* edges are followed by an *author* edge, the bold capital *B* designates the path "*/B/A*" (*book* and *author*), instead of "*/B*" alone.

A major contribution of the Index Fabric is its layered-based paging strategy to index large data. This feature makes it possible to handle very large indexes. The index structure is stored on disk and divided into multiple blocks of approximately equal size, each of which holds a small sub-Trie. The Tries of the lower levels are referenced by higher level Tries in the Index Fabric, and so forth until we reach the root Trie, which can fit in one block. The number of the Index Fabric levels is based on the size of the original data.

Index Fabric is conceptually similar to strong DataGuide [130] [32]. It is deterministic and its size may grow exponentially in the size of the original data for the DAG data, and linearly for the tree-shaped data. Furthermore, it is complete and precise for path queries, and complete for twig queries but not precise. DAG data can be indexed by an Index Fabric, but Index Fabric is more efficient for tree-shaped data.



B	book
P	paper
A	author
R	reviewer

Figure 2.12 Index Fabric of the data-tree in Figure 2.2

The Index Fabric is a balanced structure tree like a B-tree. Updating an Index Fabric may include a deletion of one record and an insertion of another. The insertion may cause one block per level of the tree to split in the worst case.

2.2.3.2 Non-Deterministic Graph Indexes with Backward Bisimilarity

The 1-index, the A(k)-index, and the D(k)-index are based on backward bisimilarity partitioning. While the 1-index backward bisimilarity length is equal to the length of the longest path in the data-graph, the A(k)-index and the D(k)-index backward bisimilarity lengths are set by a value *k*. The *k* value in the A(k)-index is set manually, and the *k* value in the D(k)-index is set automatically.

(1-index)

Milo and Suciu [90] propose 1-index as an attempt to reduce the size of a structural summary to less than that of a strong DataGuide by relaxing the determinism constraint. Figure 2.11(C) is an example of 1-index for the data in Figure 2.11(A). The 1-index partitions the data nodes of a document into equivalence classes based on their backward bisimilarity from the root node to the indexed node. Both strong DataGuide and 1-index are identical in the case of XML data-trees. In the case of DAG data, however, a 1-index may contain similar root paths, but represents each node in the source data-graph only once, and hence it is possible for a node to be reachable by multiple paths (see nodes "3" and "9" in Figure 2.11(C) for example). Based on this fact, we can say that the 1-index scheme is non-deterministic in nature. In the worst case, the size of 1-index will never exceed the size of the original data regardless of whether the data source is a basic tree or a graph. Nevertheless, 1-index structural summaries are often too large, and are considered inefficient when the original source data is large [28] and irregular.

While a 1-index represents every value in the source data only once in the index tree, a strong DataGuide may have the same value in the source data repeated in more than one location in the index tree. Hence, a 1-index is more *node centric* in its partition. Inversely, similar paths in the source data could be represented by multiple similar paths in 1-index scheme, while strong DataGuide represents all similar paths in the source data by only one path in the index. Therefore, strong DataGuide is more *path centric* in its partition.

It is easy to see from Figure 2.11(C) that 1-index is complete and precise for evaluating path queries such as "*/B/A*" and "*//R*", and is complete but not precise for evaluating twig queries like *"/P[/A]/R"*. In General, 1-index is always complete, but not necessarily precise [70].

Kaushik et al. [68] review two kinds of updates for the 1-index, namely, the addition of a subgraph, and the addition of an edge. Let the data-graph before the addition of the new file be *G*, the 1-index be *IG*, *H* is a new subgraph, and the 1-index for *H* be *IH*. Let the number of nodes in *IG*, *H*, and *IH* be *nIG*, *nH*, and *nIH*, respectively, and the number of edges be *mIG*, *mH*, and *mIH*, respectively. The time taken by the subgraph addition is $O(mHlog(nH) + (mIH + mIG)log(nIH + nIG))$. Note that this is independent of the size of *G*, but dependent on the size of *IG*, which is usually smaller than the size of the data-graph.

The complexity of edge addition is measured by the number of nodes and edges touched in the data-graph during the update process, which can be $O(n + m)$ in the worst case scenario, where *n* is the number of nodes and *m* is the number of edges in the data-graph[68].

A(k)-index

The dominant disadvantage of strong DataGuide and 1-index is the size of their indexes when the source data is large and irregular. A(k)-index is proposed by Kaushik et al. [70], mainly to overcome the size problem. Similar to 1-index, A(k)-index (Figure 2.11 (D and E)) is based on backward bisimilarity and is non-deterministic. A(k)-index uses a mechanism to minimize the size of the graph indexes by specifying a factor *k* that is used to decide the length of the backward bisimilarity of the indexed nodes. Two nodes are backward *k*-bisimilar if they share the same incoming paths of a length = *k*. For example, an

A(3)-index is an index for nodes that share the same incoming labeled (tagged) paths of length three.

The size of an A(k)-index is generally smaller than that of a strong DataGuide and a 1-index. Similar to the 1-index scheme, A(k)-index grows linearly in the size of the source data regardless of the shape of the data. A smaller value of *k* results in a smaller index. A(k)-index gains the advantage of having a smaller size at the expense of precision since the index does not necessarily reflect the complete path from the root node.

Since the A(k)-index is based on equivalence-class partitioning of nodes in a data-graph, it is usually complete but not necessarily precise [70]. We take an A(1)-index for the data in Figure 2.11(A), which is illustrated in Figure 2.11(D), as an example. For path queries such as "*//R*", A(1)- index is complete and precise as it will return the node set {3,8,9}. Although, it is complete for the path queries such as "*/B/A*", as it will return {3,9}, which is a superset of the correct answer {3}, it is not precise as the answer set contains the wrong answer "9." It is only precise for path queries with a length that is less than or equal to the length set by the *k* value. For example, an A(2)-index, as illustrated in Figure 2.11(E), is complete and precise for both "*/B/A*" and "*//R*" queries. Note that Figure 2.11(E) is identical to the 1-index in Figure 2.11(C). Actually, a 1-index is a special case of A(k)-index where *k* value is equal to the depth of a data-graph (the longest path in a graph). A(k)-index is complete but not precise for twig queries like "*/P[/A]/R.*"

The subgraph addition algorithm for the 1-index extends to the A(k)-index. Unfortunately, the edge insertion algorithm does not extend and hence the edge insertion for the A(k)-index remains an open problem [68].

D(k)-index

Choosing the correct value of *k* in the A(k)-index scheme is the biggest challenge. Large values may create a larger size index that may negatively affect the query processing for both short path and long path queries. Low *k* values, on the other hands, may produce smaller indexes and thus more efficient, but less precise, query processing. Chen et al. [28] propose D(k)-index to choose the most suitable value of *k* dynamically based on the workload. Therefore, D(k)-index is more efficient than A(k)-index with regard to processing time and storage space. In general, with regard to the rest of the above listed evaluation criteria, both D(k)-index and A(k)-index schemes share the same levels of precision, completeness, and scalability. For both D(k)-index and A(k)-index, if the length of the path in a query is longer than the value of *k*, then a post-evaluation step might be necessary to double check the correctness of the answer, which may be costly.

The D(k)-index is considered for two types of updates: the addition of a new file (subgraph), and the addition of a new edge. The update algorithm for a subgraph addition is based on the update algorithm of 1-index by Kaushik et al. [68]. On the other hand, the edge addition algorithm is novel and in general performs better than the one presented by Kaushik et al. Assume that a new edge is added to the D(k)-index *IG* from *X* to *Y*, and *Y* 's local similarity (identical structure) is equal to *Ky*. While the Kaushik algorithm, in the worst case, needs to touch *O(n+m)* nodes and edges in the data-graph, the update algorithm for the edge addition with the D(k)-index can touch nodes and edges in a distance less than or equal to *Ky* in the index graph *IG* [28].

2.2.3.3 Non-Deterministic Graph Indexes with Forward and Backward Bisimilarity

We review three types of indexing schemes under this class of graph indexes: the F&B-index, the $(F+B)^k$-index, and the disk based F&B-index. They are non-deterministic like the above type of graph indexes (1-index, A(k)-index, and D(k)-index), but they differ with respect to size and query answering power as they are larger and they cover twig queries as well as single path queries.

F&B-index

The F&B-index was introduced by Abiteboul et al. [2]. Unlike 1-index, A(k)-index, and D(k)-index which are based only on the incoming (backward) paths bisimilarity, this index scheme is based on the incoming and the outgoing (forward and backward) paths bisimilarity of all nodes in the source data-tree or data-graph. Therefore, it is considered to be a twig structural index scheme. It can be used as a covering index for the set of all branching path queries that can be expressed over a tree or graph of data.

To demonstrate the benefits of this indexing scheme, consider the twig query "*/P[/A]/R*." Evaluating this query over strong DataGuide (Figure 2.11(B)), 1-index (Figure 2.11(C)), or A(2)-index (Figure 2.11(E)), returns a set of *R* nodes {3,8,9}. We see that *R* node "3" does not contribute to the correct answer, yet it is returned in the initial steps by all the indexes. Eventually, it is eliminated from the final answer after performing some additional join steps. In contrast, as illustrated in Figure 2.11(F), the F&B-index detects this mismatch early and is able to exclude *R* node "3," therefore avoiding the additional joins and improving efficiency. F&B-index therefore is complete and precise for twig queries as well as for path queries.

The F&B-index is non-deterministic. The size of the index grows linearly in the size of the source data document, and in the worst case does not exceed the original data size for both data shapes (tree and graph). Kaushik et al. [67] proved that F&B-index is the smallest index covering all branches of a given XML graph. However, the size of an F&B-index is often too large to fit in memory. To update the F&B-index when a subgraph or an edge is added to the data-graph, approaches similar to those used for updating the 1-Index by Kaushik et al. [68] can be adopted.

(F+B)^k-index

Kaushik et al. [67] propose (F+B)^k-index, which is a modified version of the F&B-index. They manage the size of the F&B-index by specifying the value of *k* [56]. A low value of *k* results in an index that can cover limited classes of branching path queries, but the index size is often small. A high value of *k*, on the other hand, can cover a wide range of classes of branching path queries at the expense of the size since the size of the index is often large. With regard to the rest of the comparison criteria, both F&B-index and (F+B)^k-index have the same features. The idea of (F+B)^k-index as an extension to F&B-index is analogous to A(k)-index as an extension to 1-index.

Disk-based F&B-index

The main shortcoming of the F&B-index and the (F+B)^k-index is often their large sizes, because they have more details about each node. They, therefore, often do not fit in memory. To overcome this weakness, Wang et al. [131]

proposed a disk-based F&B-index with various clustering properties and criteria. They integrate 1-index with F&B-index in a new clustered disk-based F&B-index and store the index on the disk which can be dealt with efficiently as needed. In this indexing scheme, only relevant chunks of the index are returned from disk to main memory in order to be processed, which is similar to paging utilities that are available in some other indexing approaches (e.g. Index Fabric).

With regard to the other comparison criteria, in general, the disk-based F&B-index has the same characteristics and features as the regular F&B-index, in addition to the improvement in dealing with large size data. The authors of disk-based F&B-index [131] did not discuss or present any updating algorithm for their indexing scheme.

2.2.3.4 Summary of Graph Indexes

Note that 1-index and strong DataGuide indexes are suitable for small to medium size data while disk-based F&B-index and Index Fabric are more appropriate for very large data sources. Both 1-index and F&B-index are considered to be exact indexes. While A(k)-index and D(k)-index could be approximate indexes if the value of *k* for the used indexes is smaller than the length of the query path. Moreover, 1-index, A(k)-index, and D(k)-index are based on backward bisimilarity and they cover all single path queries. F&B-index and disk-based F&B-index, on the other hand, are based on forward and backward bisimilarity and they cover all branching queries for a given data set.

Table 2.2 contains a summary of the graph indexing schemes. The initial size (when it is first created) of a graph index for both tree-shaped and graph-shaped data could be either the same as the size of the data or exponential in the size of the data, in the worst case. The scalability (growing size) could be either linear or exponential in the size of data. The type of queries that are supported efficiently could be path, twig, or both.

Non-deterministic forward and backward bisimilar indexes are the only type of graph indexes that are capable of supporting twig queries if the index is exact (i.e. F&B-index). Note that the size of a deterministic index grows linearly in the original size of the source data if the shape of the source data is tree, and it grows exponentially if the shape of the source data is graph.

Table 2.2 Comparison among the three categories of graph indexing approaches

		Deterministic	**Non-deterministic Backward Bisimilar**	**Non-deterministic Forward & Backward Bisimilar**
Criteria		**Strong DataGuide, Index Fabric Approximate DataGuide**	**1-index, A(k)-index, D(k)-index**	**F&B-index, (F+B)^k-index, Disk-based F&B-index**
1-Precision	**Path**	Precise	Precise	Precise
	Twig	Not Precise	Not Precise	Precise
2-Recall	**Path**	Complete	Complete	Complete
	Twig	Complete	Complete	Complete
3- Complexity (joins required)	**Path**	No	No	No
	Twig	Yes	Yes	No
4- Size (initial, worst)	**Tree**	Same	Same	Same
	Graph	Exponential	Same	Same
4- Size (scalability, growing)		Linearly (for tree data), Exponentially (for cyclic data)	Linearly	Linearly
5- Query supported <u>efficiently</u> (without joins)		Path	Path	Path (Twig by F&B-index and disk-based F&B-index)
6- Maintain ability (Edge insertion, worst)		O (n + m)	O (n + m)	O (n + m)
Notes			- Path queries are precise for $k \geq$ path length - Edge addition to A(k)-index is not available (open for research)	- Precision and the need for joins depend on "k" value for (F+B)^k-index - Maintainability of disk-based is not available (open for research)

Before moving into the third type of structural indexes, it is worth mentioning here that graph indexes, in addition to being used as structural path summaries, can facilitate use of statistics and other features that can aid query processing and optimization [4]. For example, a graph index can hold sample values for each node or statistics about the extended data such as fan-in and fan-out of each node.

2.2.4 Sequence Indexing Schemes

Sequence indexes [104] [130] transform XML documents and queries into structure-encoded sequences. Answering a query requires sequence string matching between the encoded sequences of the data and the query. This eliminates the need for joins to evaluate twig queries. We must be careful, however, when using matching to answer a query since the sequence may not necessarily reflect a structural tree match (see Computational Complexity, Refinement Steps, next page). Sequence indexes combine the structure and the values of XML data into an integrated index structure. They are used to efficiently evaluate path queries and twig queries with keyword components without any extra join operations with tables that hold the values.

2.2.4.1 Specific Comparison Criteria of Sequence Indexes

In addition to the general comparison criteria listed above, we include the following specific comparison criteria for this type of index:

- *Computational complexity (indexing direction)*: The shape of an XML graph is similar to a triangle. At the top there is only one root element and at the bottom there may be many leaf nodes, which are usually value nodes.

A top-down search for a value in a data-tree starts from the root element then goes down the tree according to a given query path specification. In contrast, a bottom-up approach starts the search from the values at the leaf nodes. Since the selectivity of leaf nodes is higher than that of nodes in the top and the middle of the tree, a bottom-up search results in fewer paths in the tree being examined. Therefore, the indexing direction has an effect on the efficiency of a query evaluation.

- *Computational Complexity (Refinement Steps):* Sequence schemes suffer from two anomalies, namely, false positives (a.k.a. false alarms or imprecise result) and false negatives (a.k.a. false dismissals or incomplete result). Refinement steps are added to the evaluation process of a query to overcome these problems. On the one hand, the fact that these anomalies exist in the encoded sequence is an issue by itself. On the other hand, the way that these anomalies are dealt with is another issue. With regard to this criterion, we are only concerned with how efficiently these problems are resolved.

Based on the importance of tree mapping direction, we divide sequence indexes into two types, namely, top-down sequence indexing schemes and bottom-up sequence indexing schemes. ViST and PRIX are examples of top-down and bottom-up indexes, respectively.

2.2.4.2 Top-down Sequence Indexes (ViST)

The ViST (Virtual Suffix Tree) index structure is proposed by Wang et al. [130]. Before we illustrate an example of ViST, please note that the data-tree in

Figure 2.13 (B) is an encoded form of the data-tree in Figure 2.13 (A) by substituting the edge labels *Bib File*, *book*, *author*, *paper*, and *reviewer* with the letters *F*, *B*, *A*, *P*, and *R*, respectively. Furthermore, Figure 2.13 (A) is the same as the example edge-labeled data-tree in Figure 2.2. As an example of ViST, consider the data-tree in Figure 2.13 (B) and the query tree in Figure 2.13 (D). Both trees are transformed into structure-encoded sequences as illustrated below. Note that each pair in the sequence consists of the node's tag and the root path of the node's parent.

Data tree 2 (*D2*) : (F,0) (B,F) (A,FB) (P,F) (P,F) (A,FP) (P,F) (R,FP) (A,FP)
Query (*Q2.4*) : (F,0) (P,F) (R,FP) (A,FP)

The underlined subsequences of data *D2* match the query sequence of query *Q2.4*, so we return the matched subsequence in the data-tree as an answer to the query. We should be aware of any existing false positives in the solution. For example, consider the data-tree 3 in Figure 2.13 (C), the sequence of this tree is illustrated below.

Data tree 3 (*D3*) : (F,0) (P,F) (R,FP) (P,F) (A,FP)

To evaluate the above query *Q2.4* over the data *D3* data, we notice that the underlined sequence forms an answer for the query. It is not a correct answer, however, because the *R* and the *A* nodes do not have the same parent *P* node. This is an example of a false-positive answer.

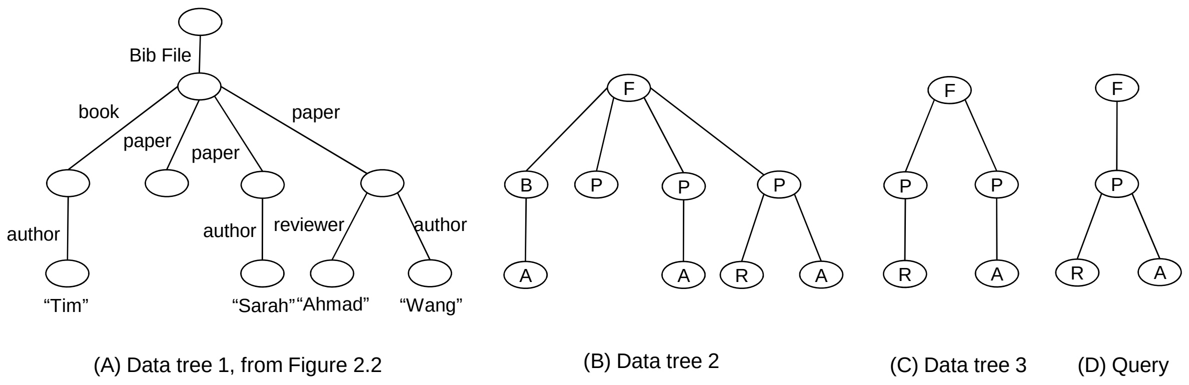


Figure 2.13 Data trees and a query

In addition to false positives, the sequence schemes also have the problem of false negatives, which is caused by the isomorphic tree problem. It occurs when a branch node has multiple identical child nodes. For example, the two tree combinations which are illustrated in Figure 2.14, have the following structural sequences.

Data tree 1	:	(F,0) (P,F) (A,FP) (P,F) (R,FP)
Data tree 2	:	(F,0) (P,F) (R,FP) (P,F) (A,FP)

If we run any one of these two trees as a query over the other tree, we will not find a match as can be seen from the translated sequences. However, logically both trees have the same structure and same number and types of elements. To solve this problem in ViST, which occurs when there are similar tag siblings in a query, we have to rewrite the given query into all possible combinations of sequence order. After that, we evaluate each query separately, and then union the result of all queries. In the worst case, permutations of the query sequence are exponential in the number of the similar siblings.

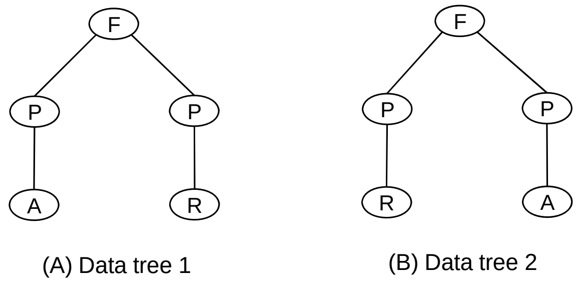


Figure 2.14 An example of false-negative

ViST is based on top-down traversal tree. As a result, for deep and large XML documents, the size of the index becomes a problem as it does not scale well with an increase in data size because the top elements have to be included within the sequence of the newly inserted elements. As the paths in XML data get longer, the sequence length will increase and hence the size of the index will increase exponentially in the size of data.

The false positives problem is resolved by disassembling the query tree at the branch into multiple trees, and using join operations to combine their result. This solution is definitely expensive, since it involves additional join operations. ViST, which is based on the B+-tree [130], is physically implemented as two levels of B+-trees [56]. If we assume that the fan-out of the used B+-tree is equal to b, then $O(b \log_b n)$ nodes are touched during a sequence index update at each level, where n is equal to the number of nodes in the data-tree.

2.2.4.3 Bottom-up Sequence Indexes (PRIX)

ViST's top-down transformation approach weakens the query processing because it results in a large number of nodes (paths) being examined during subsequence matching for commonly occurring non-contiguous tag names.

Motivated by this fact, Rao and Moon [104] propose another approach that implements bottom-up transformation instead. This approach is called PRIX (PRufer sequences for Indexing XML). It is based on Prufer sequences as indicated by the name. The bottom-up transformation of XML data-trees in PRIX plays a crucial role in reducing the query processing time.

Basically, the top-level elements of an XML tree are shared with lower-level elements by being their parent or ancestor nodes. Thus, if we index a tree starting from the top, the chances are high of having a large number of elements that share the same starting tags in a given query path. In contrast, indexing a tree starting from the bottom and moving upward to the top of the tree reduces the chance of having a large number of shared elements for a given query path as the selectivity is higher at the bottom. A bottom-up index is more efficient than a top-down index and PRIX therefore is more efficient than ViST [104].

PRIX is based on Prufer sequences. To illustrate how a Prufer sequence is used to denote a tree, we use the data-tree in Figure 2.15, which is the same as the data-tree in Figure 2.13 (B). The letters inside the node circles represent the tag types (labels) and the numbers shown beside the nodes represent the post-order numbering of the tree. To encode the tree in Figure 2.15 with a Prufer sequence, we repeatedly delete the leaf node that has the smallest number and append the label of its parent to the sequence.

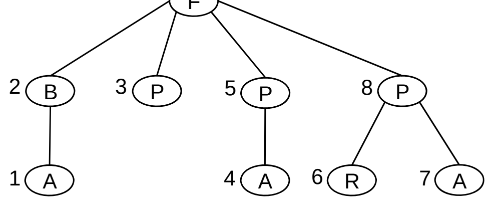


Figure 2.15 An example of Prufer sequence

As we can see in Figure 2.15, the smallest post-order number is “1” so we delete it and add “2” to the sequence, so it becomes {2}. We delete the node numbered “2” and add its parent “9” to the sequence to become {2,9}, and so forth. At the end of this process, we have the sequence {2,9,9,5,9,8,8,9}, which represents the following tag sequence {*B,F,F,P,F,P,P,F*}.

In PRIX the string/character data in the XML document tree are extended by adding dummy child nodes before the transformation process so it can be indexed using the Prufer sequence. Similarly, query twigs are also extended before transforming them into sequences. Indexing extended Prufer sequences is useful for processing twig queries with values. Since queries with value nodes usually have high selectivity, they are processed more efficiently than those without values.

The size of a PRIX grows linearly in the total length of the sequences stored in it because an increase in the path length will result in a sequence addition which is equal to the amount of the increase. In the PRIX approach, the length of a Prufer sequence, as we noticed from the above example, is linear in the number of nodes in the tree. Hence, the index size is linear in the total number of tree nodes regardless of the depth of the tree.

PRIX uses a complex four-phase refinement process to deal with false positives and false negatives. Basically, PRIX overcomes the false positives problem by using document by document post-processing which is a time consuming process. PRIX is based on the B+-tree, and it is built in a way similar to ViST [130]. It is mainly implemented as two levels of B+-trees. If we assume that the fan-out of the used B+-tree is equal to b, then $O(b \log_b n)$ nodes are touched during a sequence index update at each level, where n is equal to the number of nodes in the data-tree.

2.2.4.4 Summary of Sequence Indexes

Table 2.3 includes a summary of the sequence indexing schemes. Indexing can be implemented in either a top-down direction or bottom-up direction. Both single path and twig queries are supported efficiently by sequence indexes.

Table 2.3 Comparison between Top-down (ViST) and Bottom-up (PRIX) sequencing schemes.

No	Criteria		Top-down (ViST)	Bottom-up (PRIX)
1	**Precision**		False-positives (imprecise)	False-positives (imprecise)
2	**Recall**		False-negatives (incomplete)	False-negatives (incomplete)
3	**Computation Complexity**	**Refinement step**	Expensive Joins	Complicated four-phase process
		Indexing direction	Top-down	Bottom-up
4	**Scaling/Size**		Exponential	Linear
5	**Type of queries supported <u>efficiently</u> (without joins)**		Path & Twig	Path & Twig
6	**Maintainability**		$O(b\log_b n)$	$O(b\log_b n)$

2.2.5 Structural Indexes Critique

As is always the case with indexing schemes, there is a trade-off between the size and the precision of the index on the one hand, and between the size and the efficiency of the index in answering a query on the other hand [110]. The advantages of one index scheme can be the disadvantages of another. In this section we compare the three categories of structural indexes, namely, node index schemes, graph index schemes, and sequence index schemes.

2.2.5.1 Criteria for Comparison among Structural Indexing Schemes

In addition to the general criteria listed previously we use the following specific criteria to compare the above three types of structural indexing schemes:

- *(A) Computational complexity: Does it require structural joins?*

  Structural joins are considered for path queries and twig queries. In general, to achieve high performance for a query execution, we need to minimize the number of joins.

  (B) Computational complexity: Granularity of usage to evaluate a query.

  The granularity of an XML index depends on the type of the indexing scheme. For example, the granularity could be at the node level, the path level, or the twig level (for twig queries). As the granularity of the index that is used to evaluate a query become coarser, the execution time becomes shorter.
- *Data supported.*

  The types of data supported by the XML indexing schemes are mainly tree-shaped data and graph-shaped data. The main difference between them is that the graph-shaped data can be used to represent an XML document with the ID/IDREF attribute tokens. The tree-shaped data can be considered as a subclass of the graph-shaped data where a node cannot have more than one parent. The indexing schemes that are capable of supporting the graph-shaped data are more powerful than the ones that support only the tree-shaped data.

- *Ability to facilitate the use of statistics and other features.*

  The ability to facilitate the use of statistics, such as the fan-in and the fan-out of nodes, helps to provide query optimization with the capability to choose the most efficient evaluation plan for a given query.

- *Values integrated into the index structure.*

  If the values of the elements and attributes are indexed separately from the structure, and a query with some predicates needs to be evaluated over that data, then joins between the structural index and the value indexes are necessary and hence increases the complexity of the XML query evaluation process. In contrast, integrating values into the structural index saves some additional joins and narrows down the matching procedure during the evaluation process, since the selectivity of the values are always higher than that of the elements in a structural index.

2.2.5.2 Comparison among Structural Indexes

Generally, sequence indexes may initially produce a wrong answer to a query then correct it at a later stage in the evaluation process. The deterministic graph indexes and non-deterministic graph indexes with backward bisimilarity may produce some wrong initial answers. The non-deterministic graph indexes that are based on forward and backward bisimilarity, on the contrary, are more accurate and often return only the correct answers. Finally, since the node indexes are used for binary joins, they do not produce any initial wrong answers.

Without some extra post-processing steps, false negatives may occur when we use a sequence indexing scheme to evaluate a query. On the other hand, node and graph indexes always return a complete answer because the order of the nodes is not encoded within the structure of the index.

The number of structural joins that are required to evaluate a path or a twig query varies among the different schemes. It has a significant impact on the query processing time. Node indexes are the least efficient with respect to structural joins since they require joins for both single path and twig queries. Graph indexes support single path queries without the need for structural joins but structural joins are required (for all graph indexes except F&B-index) at the branching node to evaluate twig queries. Finally, sequence indexes are the best because the structure is encoded within the sequence so they do not require any structural joins for single path or twig queries.

There are three levels of granularity used to evaluate a twig query: the pair-wise, path, and twig levels. For illustration, in order to evaluate a twig query using a node index, we break the query into nodes, then join nodes a pair at a time until all nodes are joined together for the complete twig path to evaluate the query. On the other hand, to evaluate a twig query using all graph indexes except F&B-index, we break the query into several singular paths and evaluate each path separately, then join the results of all paths to form the answer to the query. Finally, to evaluate a twig query by using a sequence index, we process the twig query as a whole.

Node indexes can only support tree-shape data because of the containment rule that is used to specify the relationship between two nodes in a data-tree. In order for node A to be an ancestor of node B, A's interval code has to contain B's interval code, and not vice versa, which may be caused by a graph-shaped data.

In contrast, graph indexes support the graph-shaped data well. Like node indexes, sequences indexes only support tree-shaped data.

Some indexes provide valuable assistance for query optimization. For example, strong DataGuides [54] are used in Lore [87] to facilitate annotation of sample values and statistical data. The annotated information is associated with the DataGuide objects (nodes). This information assists in estimating the cost of the evaluation plans for a given query. The node and the sequence indexes do not facilitate these kinds of supporting information.

There are some attempts to integrate values into graph indexes [37] [133], although, the majority of graph indexes do not carry any values within the structural summary. Node indexes cannot contain values, and values have to be indexed separately. The only indexing schemes that are designed to efficiently integrate values into the structural index are the sequence indexing schemes. We observe that node indexes are mainly used for path joining, graph indexes for path selection, and sequence indexes for complete query evaluation.

We summarize our comparison of the three categories of structural indexing schemes in Table 2.4. The granularity of usage to evaluate a query could be at the node level, the path level, or the twig level. The types of queries that are supported efficiently without joins by these indexing schemes could be path, twig, or both. The maintainability of graph and sequence indexes is measured by the number of nodes that are needed to be touched during the update process. On the other hand, the maintainability of node indexes are measured by the size of used labels. The supported data could be a tree-shaped or a graph-shaped. Tree-shaped data is considered a subset of graph-shaped data.

Table 2.4 Summary of comparison among the 3 categories of structural indexing schemes.

Criteria		Node Indexes	Graph Indexes	Sequence Indexes
1- Precision (wrong initial answer, false positive)		No	Yes/No	Yes
2- Recall (missing initially correct answer, false negative)		No	No	Yes
3- Computation complexity (structural join required)	**Path**	Yes	No	No
	Twig	Yes	Yes / No	No
3- Computation complexity (granularity of usage to evaluate a query)		Nodes Pair-wised Evaluation	Path Evaluation	Twig Evaluation
4- Size / Scalability		Linear-Exponential	Linear-Exponential	Linear-Exponential
5- Type of queries supported <u>efficiently</u> (without joins)		None	Path (Twig by exact (F&B) indexes)	Path & Twig
6- Maintainability for adding an edge		O(n) immutable $O(\log n)$ mutable	$O(n + m)$	$O(b\log_b n)$
7- Data supported		Tree	Graph	Tree
8- Can facilitate the use of statistics		No	Yes	No
9- Hold value		No	Yes/No	Yes

2.3 Summary

Indexing is key factor in improving the performance of XML queries [146]. Indexes are used during most of the optimization stages. Indexing the XML data has to reflect the structure in order to be able to support XML queries. An XML query consists of two parts: (1) the structural part, which is specified by the components' structure of the query; (2) the values that are associated with these components.

Our classification of XML graph indexes is novel. It is based on their deterministic property in addition to forward and backward bisimilarity, which determines the possible size and accuracy of an index. Deterministic indexes may grow exponentially in the worst case, while non-deterministic indexes grow linearly. Forward and backward bisimilar indexes are more accurate than backward bisimilar indexes. Deterministic indexes guarantee uniqueness of paths, and are suitable for single path queries. They evaluate a single path query by traversing one path only. In contrast, non-deterministic graph indexes may traverse more than one index path to evaluate a single path query. Our classification of XML sequence indexes is also novel. It is based on the mapping direction of data-trees, because the mapping direction is the main factor that drastically affects the size of sequence indexes and their answering power. The best way to judge the strength of an indexing technique is to compare it with other techniques using common criteria that are applicable for all of them and can act as a benchmark. In this chapter, we use common criteria to analyze the characteristics of the most common types of XML structural indexes.

Our analysis of structural indexes is based on the following key issues: retrieval power, which covers the precision and the completeness of an index; processing complexity, which demonstrates how efficient an index can be used to answer a query; scalability of the index and its adaptability to queries with different path lengths; and finally update cost of the index.

We observe that no single indexing scheme is capable of satisfying all users' needs; deciding which index scheme to use depends on the users' preferences [22]. There is a trade-off between the size of the structural index and its precision. For example, graph indexes with only backward bisimilarity tend to have lower accuracy (which is corrected by some post-processing steps) but their sizes are minimal. In contrast, graph indexes with forward and backward bisimilarity tend to have high accuracy but at the expense of the size. Node and sequence indexes can be used only for tree-shaped data, while graph indexes can be used for both tree-shaped and graph-shaped data. Graph indexes can be used to efficiently facilitate additional information such as some statistical information [139], which can be used during a query optimization process. Some indexes cover twig and single path queries, while others cover only single path queries.

Finally, the ultimate goal of researchers is to create an indexing scheme that will occupy minimal storage without compromising the precision, if possible, or at least improve the trade-off in favor of precision (i.e. have a small increase in the size to achieve higher precision).

Chapter 3

LLS: Level-based Labeling Scheme for XML Databases

Labeling nodes of XML trees to reflect the structure is useful for indexing and retrieving XML data. Current labeling schemes can be divided into two groups: interval labeling and prefix labeling schemes. In this chapter, we first discuss the advantages and disadvantages of the two groups. We then propose a novel labeling scheme, Level-based Labeling Scheme (LLS), which has the advantages of the two types of schemes while eliminating the main disadvantages. The LLS is based on the levels of the nodes in XML trees and the summary of an XML tree. We provide a set of experiments that indicate the performance benefits of our proposed scheme compared with interval labeling schemes using different mappings to relational tables to implement the indices. We end the chapter by summarizing our contributions

3.1 XML Labeling Schemes

Node indexes depend on the labeling schemes used. In this section, we discuss the most popular types of labeling schemes. We then discuss their importance in XML data storage and retrieval. Finally, we discuss their advantages and disadvantages, which motivate us to propose LLS. The labeling schemes are discussed in depth in Chapter 2. For the purpose of suggesting a novel labeling scheme that combines two types of labeling schemes in one integrated labeling scheme and for ease of reference, we review in this chapter the information that is related to the integrated scheme.

3.1.1 Types of Labeling Schemes

Node indexes hold values that reflect the nodes' positions within the structure of an XML tree. Node indexes depend on labeling schemes [7]. Two of the most widely used types of schemes are interval (a.k.a. region) labeling and prefix (a.k.a. path) labeling.

The (*Beg,End*) labeling scheme proposed by Zhang et al. [143] is an early interval labeling scheme. In this scheme, each node in an XML tree is given a *beginning* and *ending* number based on the sequential traversal of XML document. Two nodes are related if one of the node's interval contains the other node's interval. Figure 3.2 is an example of the (*Beg,End*) labeling scheme for the data in Figure 3.1.

```
<Bib>
    <book>
        <author>Tim</author>
    </book>
    <paper> </paper>
    <paper>
        <author>Sarah</author>
    </paper>
    <paper reviewer="Ahmad">
        <author>Wang</author>
    </paper>
</Bib>
```

Figure 3.1 XML document

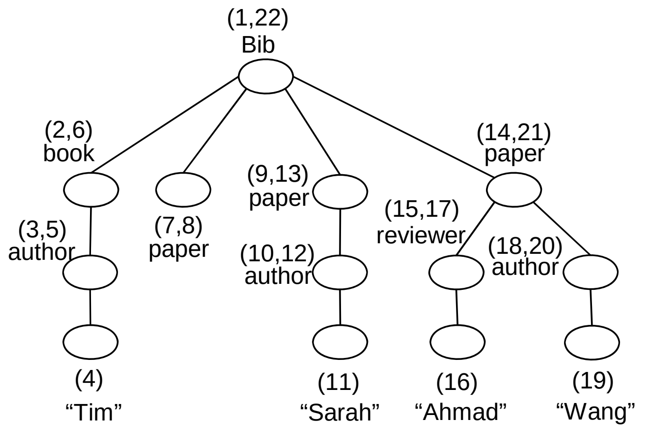


Figure 3.2 A (*Beg,End*) labeled tree representation of the XML document in Figure 3.1

Li and Moon [78] propose another type of interval labeling scheme called the (*Order,Size*) labeling scheme. The *Order* part is based on a pre-order traversal, and the *Size* part is an estimate of the number of the child-descendent nodes for a given node. This durable approach may provide intervals for a certain number of new nodes equal to the gap size in order to avoid relabeling of the data-tree nodes in case of insertion. In this case, relabeling may be delayed, but eventually it is required. It occurs more often if the data distribution in the tree is skewed.

Amagasa et al. [8] use real numbers (rational numbers) instead of integers to represent an interval in node indexing to further delay relabeling on nodes upon insertion. Wu et al. [135] propose a novel labeling scheme that uses prime numbers to label nodes in an XML tree. In this approach, each node's label can only be divided exactly by its own ancestor(s).

The *Dewey code* labeling scheme proposed by Tatarinov et al. [119] is an early prefix labeling scheme. In this scheme, each node is associated with a vector of numbers that represents the node-ID path from the root to the designated node. To decide if there is a relation between two nodes, we perform a prefix matching operation on the nodes index strings. Figure 3.3 is an example of the Dewey code labeling scheme for the data in Figure 3.1.

O'Neil et al. [99] propose the *ORDPATH* labeling scheme that is similar to the Dewey code labeling scheme, except that the child nodes of a given parent node are labeled by using odd numbers, and even numbers are used later for new insertions. This labeling scheme handles insertion gracefully. Fisher et al. [48] propose a dynamic labeling approach that can be applied to Dewey code labels when there is type information in the form of DTD or Schema.

Other prefix labeling for XML trees include *GRP* [80] and *LSDX* [42]. In the GRoup base Prefix (GRP) labeling scheme, the labels consist of two parts, namely, group ID and group prefix, while in the Labeling Scheme for Dynamic XML data (LSDX), the labels are a combination of numbers and letters.

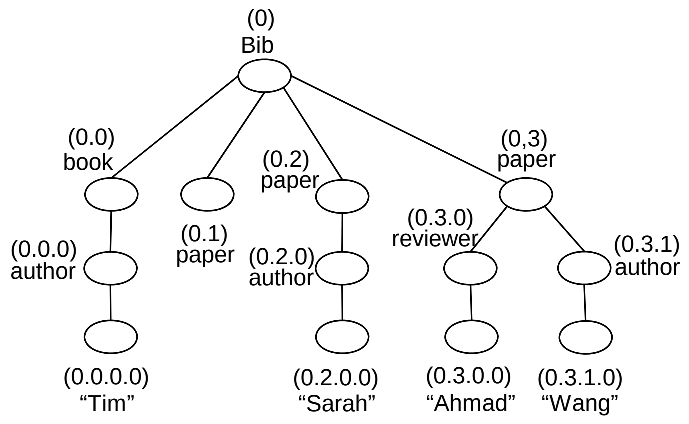


Figure 3.3 A Dewey code labeled tree representation of the XML document in Figure 3.1

3.1.2 Importance and Usage of Labeling Schemes

Node indexes [8] [48] [78] [99] [119] [135] [143] are used at the granularity of individual nodes in a query path. Graph indexes [2] [28] [37] [54] [67] [70] [90] [131] are used at the granularity of sets that represent paths in a query. Finally, Sequence indexes [104] [130] are used at the granularity of a complete query path. Node indexes depend on labeling scheme used. Unlike graph and sequence indexes that may suffer from false positives and false negatives, node indexes always return precise and complete answers [92] [93]. Node indexes require more join operations to evaluate a query in comparison to graph and sequence indexes. Nevertheless, precision and completeness are the core advantages of node indexes over other types of indexes, which explain their popularity among XML structural indexes.

Node indexes depend on the labeling scheme used. Two of the most widely used labeling schemes are (*Beg*,*End*) [143] and *Dewey code* [119], which belong to interval group and prefix group of labeling schemes, respectively. Node indexes can be used exclusively to evaluate XML queries. Graph and sequence indexes, however, need some kind of labeling scheme to label individual nodes in XML trees in order to work properly.

3.1.3 Limitations of Existing Labeling Schemes and Introduction to LLS.

Comparing the interval and prefix labeling schemes, we notice that each type's advantages are the disadvantages of the other [92] [93]. The interval labeling schemes require fixed time to compute a relationship between any two arbitrary nodes for two reasons. First, it uses numerical values to index the nodes. Second, the size of the label that is used to index each node is fixed depending on the depth of the tree. On the contrary, in prefix labeling schemes, the time that is required to compute the relationship between any two arbitrary nodes is directly proportional to the depth of the nodes for two reasons. First, prefix labeling schemes use strings to represent labels instead of numbers. Second, the labels' size increases as the depth increases [93] [56].

Unlike interval labels, each prefix label contains the root path (the path from the root to the designated node) information. Therefore, with prefix labels, we can infer any node's parent-child or ancestor-descendent from the label of the node [93].

Finally, prefix labels are often easier to update than interval labels. Updating interval labels are costly. When a new node is inserted into a data-tree, then all the nodes in the tree, except the left sibling subtree(s) of the inserted node, have to be updated [56] [93]. While in prefix labeling scheme, when a new node is inserted, only the nodes in the subtree(s) rooted at the following sibling(s) need to be updated [93] [119].

Because interval node indexes require fixed time to compute a relationship between any two arbitrary nodes, we believe that they form a solid foundation for strong node indexes. We also believe that interval node indexes can be extended to have the advantages of prefix indexes. We therefore propose the LLS labeling scheme, which is based on numerical values, has fixed-size labels regardless of the depth of the node, requires a fixed time to compute a relation between two nodes, can be used to infer the parent-child and ancestor-descendent nodes from their labels, and requires modest amount of relabeling upon insertion. Table 3.1 contains a synopsis of the characteristics of the proposed LLS labeling scheme in comparison to the interval and prefix labeling schemes.

Table 3.1 A comparison among Interval, Prefix, and LLS labeling schemes

	Interval labeling	**Prefix labeling**	**Level-based Labeling (LLS)**
Relationship computation	Fixed	Directly proportional to depth increase	Fixed
Data type	Numerical	String	Numerical
Size	Fixed	Directly proportional to depth increase	Fixed
Can infer exact related nodes	No	Yes	Yes
Maintenance cost	Expensive (may impose considerable relabeling)	Less expensive (may impose limited relabeling)	Less expensive (may impose limited relabeling)

The motivation behind the design of LLS is to have a single labeling scheme that has the advantages of both groups of labeling schemes, while avoiding their disadvantages. The LLS is capable of processing single path queries as well as twig queries. The LLS is based on the levels of the elements in XML trees. The element labels and values are tightly coupled with a structural summary so the method lends itself to efficient query processing and is shown here to perform well in comparison to existing labeling schemes.

3.2 Our Approach: LLS Labeling Scheme

In this section, we first define our basic data model that is used in the LLS. Then we discuss the updating complexity of the LLS labeling scheme. Finally, we present two simple examples to illustrate how LLS labels are used to evaluate a query in comparison with *(Beg,End)* interval labels.

3.2.1 Data and Graph Index Models

We model an XML document as a directed graph $G=(R,V_R,V_L,E,tagg,labelg,T)$. R is the root node. V_R is the set of elements and attributes (internal nodes), excluding R and V_L. $V_L = (V_T \cup V_E)$ and V_L is the set of leaf nodes that contain the set of value (text) nodes, V_T, and the set of empty elements nodes, V_E. Nodes in V_R and V_L are tagged with the *tagg* function (the extra *g* stands for the graph G). Nodes in V_R and V_E are tagged according to the tags of the elements or attributes they represent. Nodes in V_T have the same tags as their V_R parent nodes. A node v, such that $v \in V_R$, has one or more child nodes, which could be V_R and/or V_L node(s). E is a set of child-parent edges, $E=\{e_1,e_2,\ldots,e_i\}$, that connect all nodes of V_R and V_L to form a tree. The total number of edges is $|E|$ and the total number of nodes is $|V_R| + |V_L|$, where $|E| = |V_R| + |V_L|$ since $R \notin V_R$. Each node in V_R and V_L is associated with only one parent through an edge, except R, which does not have a parent since it is the root node[2].

All nodes in V_R and V_L are assigned unique labels through the *labelg* function, which is determined by the LLS labeling scheme as follows. Each node *v,* such that $v \in (V_R \cup V_E)$ is assigned a unique vector label *<d.p.s>* where *d* and *p* are taken from the label of the *o* node in the summary *S* (Figure 3.5) to which *v* node belongs according to an earlier implemented partition. That is, *v* node is an instance of an *o* node (instance and summary are defined later). *s* is the instance serial number of node *o*, which uniquely identifies this node among similar nodes of the same class. Nodes in V_T carry the same labels as their V_R parent nodes. The set of serial paths is defined by *T*, where $T=\{ r_1, r_2,\ldots,r_n\}$ and *n* is the number of leaf nodes $|V_L|$. We define serial path *r* in Definition 3.2 below. In our model, an edge *e* of a node *v*, where $e \in E$ and $v \in (V_R \cup V_E)$, is equal to the serial number *s* of the parent node *p*, denoted *e(v)=s(p)*. The edges of the nodes in V_T are equal to the edges of their parent nodes. The data-tree graph representation *G* for the data in Figure 3.1 is illustrated in Figure 3.4, which is used in the examples throughout this chapter, unless we state otherwise. Next, we give several definitions, which are used in describing the LLS labeling scheme.

Definition 3.1. A *tag path t* for a node *v* is a sequence of tags, $l_1.l_2\ldots l_i$ ($i \geq 1$), of the nodes on the path from the root node to *v* node, separated by dots. For example, the tag path of node <3.31.1> is *Bib.paper.author*.

Definition 3.2. A *serial path r* for a node *v* is a sequence of serial numbers, $s_1.s_2\ldots s_i$ ($i \geq 1$), of the nodes on the path from the root node to *v*. For example, the

[2] REF/IDREF are encoded as values in XML, and can be related through their values, hence we do not consider them as edges.

serial path of node <3.31.1> is (*1.2.1*), which contains the third part of the labels of the nodes in the path from the root node to this node. Note that the *d* values (the levels) of the components of a serial path *r* of a node *v*, where $r = (s_1.s_2 \ldots s_i)$, is $d = (1,2,\ldots,i)$, respectively, where *i* is the level of *v*. For example, for node <3.31.1>, the levels of the component of the serial path (*1.2.1*) are (1,2, and 3), respectively.

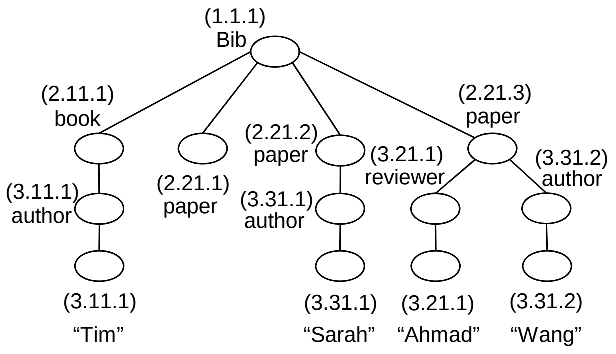


Figure 3.4 An LLS labeled tree representation of the XML document in Figure 3.1

Definition 3.3. A *node path n* for a node *v* is a sequence of alternating tags and serial numbers $l_1.s_1.l_2.s_2\ldots l_i.s_i$ ($i \geq 1$), of the nodes on the path from the root node to *v* node. For example, the node path of node <3.31.1> is *Bib.1.paper.2.author.1.*The tag path *t* of a node path *n*, denoted *t(n)*, is the sequence of tags that exist in *n*. For example, *t(n)* of *Bib.1.paper.2.author.1.* is *Bib.paper.author*. Similarly, the serial path *r* of node path *n*, denoted *r(n)*, is the sequence of serial numbers that exist in *n*. For example, *r(n)* of *Bib.1.paper.2.author.1.* is (*1.2.1*).

Definition 3.4. A node with a node path *n* is an *instance* of a tag path *t* if the sequence of the tag path in *n* is identical to the sequence of the tag path *t*, *t(n)*=*t*.

For example, the nodes <3.31.1> and <3.31.2> are instances of the tag path *Bib.paper.author*.

Definition 3.5. *Extension* of a tag path *t*, denoted *ext(t)*, is a set of nodes whose node paths are instances of a tag path *t*, that is, *ext(t)*={*n* : *t(n)*=*t* }. For example, the extensions of the tag path *Bib.paper* are nodes <2.21.1>,<2.21.2>, and <2.21.3>.

All nodes of an XML data-tree *G* can be summarized by a *summary S* such that all node paths of *G* that share the same tag path *t* are represented by exactly one tag path *t* in *S*, and every tag path *t* of *S* is a tag path of at least one node path *n* of *G*. That is, every distinct path in the source data to appear only once in the summary, and all the paths in the summary have at least one matching path in the original source data. Basically, *G* nodes are partitioned into equivalence classes in *S* where the nodes of a class have the same root path [54].

We define the *summary* as a directed graph *S=(R,O,M,tags,labels,C)*. *R* is the same as the data graph *G* root element, since an XML document can have only one root element. *O* is the set of index nodes excluding *R*. *M* is the set of child-parent edges that connects *O* nodes to form a tree. $|M|=|O|$, where $|M|$ is the total number of edges in the index tree and $|O|$ is the total number of nodes in the index tree. Nodes in *O* are tagged through the *tags* function. We tag *O* nodes with the tag name of the element or attribute they extend. All nodes in the summary are assigned a unique label through the *labels* function, which is determined by the LLS labeling scheme as follows.

Each node's label consists of a two part vector <*d.p*>, where *d* is the level (depth) of the node, and *p* is the number of this node across the *d* level (denoted as *PerLv*). An edge *m* of a node *o*, where $m \in M$ and $o \in O$, is equal to the *p* value of the parent node *x*, denoted *m(o)=p(x)*. *C* is the set of counts of instances for each node in *O*, that is, $C=\{c_1,c_2,\ldots,c_i : i=|O|\}$. For each node o_j, and count c_j, where $o_j \in O$ and $c_j \in C$, c_j is the count of instances of the tag path t_j of node o_j, where $O=\{o_1,o_2,\ldots,o_i : i=|O|\}$, $t=\{t_1,t_2,\ldots,t_i : i=|O|\}$, and node o_j has tag path t_j. If we assume that in *O* there is a node o_j whose count of instances is c_j, and c_j value is x, then the *s* values of the instances of o_j would be 1 for the first instance, 2 for the second instance, … , and *x* for the last instance. Figure 3.5 contains an example of a summary *S* of the XML data-tree *G* in Figure 3.4.

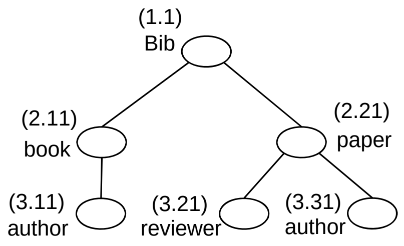


Figure 3.5 The Summary *S* of the XML data-tree *G* in Figure 3.4

For each node o_i in *S* that has a label $<d_i.p_i>$, there are instances in *G* that have labels of the form $<d_g.p_g.s_g>$, such that $d_i=d_g$, $p_i=p_g$, and $s_g=\{1,2,\ldots,n\}$ where *n* is equal to the count of instances of o_i, that is, $n=c_i$. For example, the numbers beside the oval shaped nodes in Figures 3.4 and 3.5 represent the labels of the nodes according to *labelg* and *labels* functions, respectively.

Note that the labels of the summary nodes in Figure 3.5 are created first, and then used to create the labels for the data-tree nodes in Figure 3.4. The gaps between the *PerLv* numbers in Figure 3.5 allow for expansion while maintaining the order of the elements. The gaps' intervals can be specified based on the cardinality of existing nodes.

The summary *S* information of Figure 3.5 is mapped into table representation as shown by the *Summary* table in Figure 3.6(A), and the *V*L (leaf nodes) information of the data-tree *G* of Figure 3.4 is mapped into table representation as shown by the *Values* table in Figure 3.6(B).

Tag	**Lev**	***PerLv***	***Parent***	***Type***	***Count***
Bib	1	1	0	E	1
book	2	11	1	E	1
paper	2	21	1	E	3
author	3	11	11	E	1
reviewer	3	21	21	A	1
author	3	31	21	E	2

(A) Summary table

Lev	***PerLv***	***No***	***Value***	***SerPath***
2	21	1	null	1,1
3	11	1	Tim	1,1,1
3	31	1	Sarah	1,2,1
3	21	1	Ahmad	1,3,1
3	31	2	Wang	1,3,2

(B) Values table

Figure 3.6 The Summary and Values tables of the data in Figures 3.5 and 3.4, respectively

In Figure 3.6(A), the *Tag* field contains the tag of the element of the nodes in the summary, which is assigned through the *tags* function of *S*. The *Lev* and *PerLv* fields represent the *d* and the *p* parts of the summary nodes labels as indicated in Figure 3.5, respectively. These labels are allocated through the *labels* function of *S*. The *Parent* field holds the labels of the parent nodes, which are the *p* values of the parent nodes. The *Lev*(*d*) value of the parent node is equal to the current node *Lev* value minus one, so we do not need to list the parent node level in the Summary table. Note that the *Parent* value of the root element is zero since it does not have a parent. The *Type* represents the type of node (e.g. element or attribute). The *Count* value (*C*) is the number of nodes in the original XML data that belong to the same summary group. It is used mainly to reconstruct the subtrees that are rooted at the internal nodes *V*R.

In Figure 3.6(B), the *Lev*, *PerLv*, and *No* values together form the labels of the leaf nodes <*d.p.s*>, as shown in the data-tree in Figure 3.4. These labels are allocated through the *labelg* function of *G*. The *Value* field contains either the values of the value nodes V_T, or *null* for the V_E nodes. Note that the labels of the V_T nodes in the *Values* table (which consist of *Level*, *PerLv*, and *No*) are the same as the element or attribute labels to which they belong. Finally, the *SerialPath* field contains the serial path *r* of each node in the tree. It represents a vector of the *No* values of the nodes that constitute a path from the root node to the designated node.

Note that the internal nodes V_R can be inferred and reconstructed by using the *Summary* table along the *SerPath* field in the *Values* table. We therefore do not store them. We implement the *Summary* and the *Values* tables as relational tables. The primary key fields of each table are underlined in Figure 3.6.

3.2.2 Cost of Updating the LLS Labels

When a new node is inserted into a database, it affects the corresponding index structure of the database in one of the following two ways. First, if the inserted node changes the structure of the summary (i.e. the inserted node does not belong to any of the existing paths in the summary *S*), and all the gaps are used in the summary, then the subsequent sibling nodes across the level of the inserted node have to be updated, as shown in Figure 3.7. These updates in the summary nodes have to be reflected on the data nodes too. This type of update may therefore be expensive. In order to minimize the cost, we can increase the gap between the *PerLv* numbers. Also, we can carry out the relabeling process in the direction that requires fewer nodes to be relabeled. For example, the relabeling could be carried out to the previous siblings′ nodes if it is cheaper than relabeling the following siblings′ nodes.

The second case of an update is where the inserted node does not affect the summary (i.e. the new node belongs to an existing tag path *t* in the summary *S*). In this case, only the nodes that belong to the same group and located to the right of the inserted nodes must be updated as shown in Figure 3.8. This type of update is cheaper than the first type. It involves updating the data nodes only.

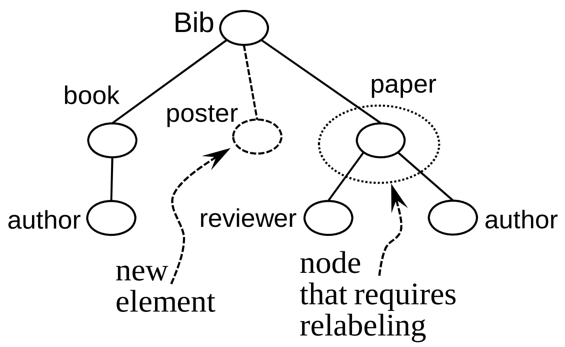


Figure 3.7 A relabeling scenario of LLS summary

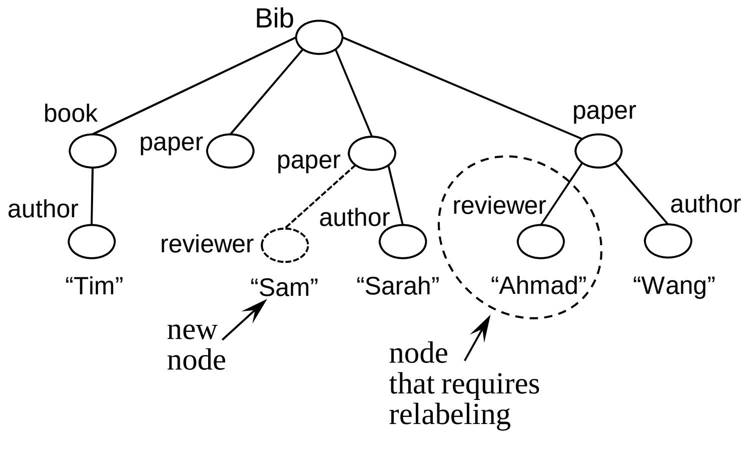


Figure 3.8 A relabeling scenario for an LLS labeled data-tree

Figure 3.9 illustrates the necessary relabeling of the nodes following the addition of a *paper* node to the XML document shown in Figure 3.1 using interval, prefix, and LLS labeling schemes. Since our labeling scheme is based on the levels of a tree where each level's nodes are labeled independently of other levels' nodes, the insertion of a node requires the relabeling of only one level's nodes in the worst case as illustrated in Figure 3.9(C). In contrast, prefix labeling may require the relabeling of more than one level of nodes, as illustrated in Figure 3.9(B). The update cost of data nodes in our approach is approximately equal to the cost incurred by the Dewey code prefix labeling scheme [119], which is less than that incurred by the *(Beg,End)* interval labeling scheme[143]. If a new node is inserted into a data-tree that is labeled with the *(Beg,End)* interval labels (see Figure 3.9(A)), then the labels of all nodes in the data-tree have to be updated, except the nodes rooted at the previous siblings of the inserted node [56], and in the worst case, the labels of all nodes in the tree are subject for relabeling. A subtree insertion is dealt with as a group of individual nodes where one node is inserted at a time.

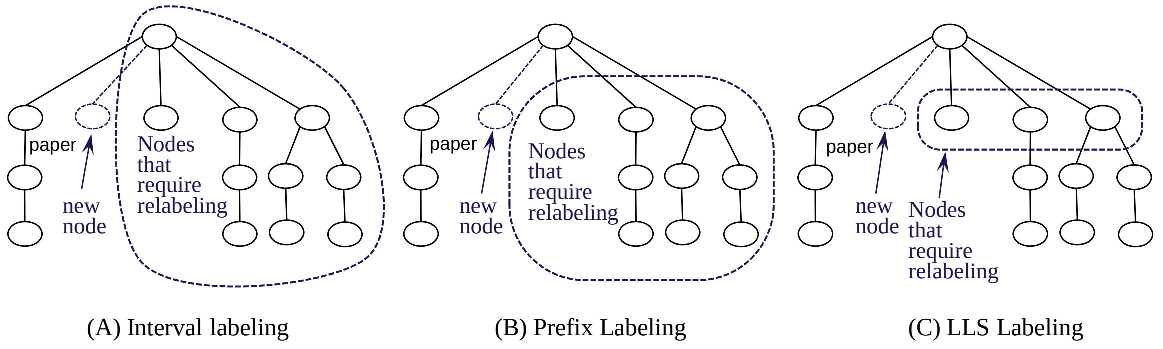


Figure 3.9 Worst case relabeling scenarios for Interval, Prefix, and LLS encoding

3.2.3 Mapping to Relational Database Tables

In this section we present two examples to illustrate how the *(Beg,End)* and the LLS are used to evaluate XML queries by using different types of mappings to relational database tables. Note that the data in Figure 3.2, which represent a *(Beg,End)* interval labeled tree, can be mapped into the two relational tables shown in Figure 3.10, where the primary key fields of the tables are underlined. This mapping is similar to the mapping suggested by Zhang et al. [143], who introduced the interval labeling to XML documents. We called this mapping the *basic mapping*. In this mapping, all nodes in the XML tree are mapped into one table, the *Nodes* table (Figure 3.10(A)); and the values are mapped into a separate table, the *Values* table (Figure 3.10(B)).

Tag	*Beg*	*End*	*Lev*
Bib	1	22	1
book	2	6	2
author	3	5	3
paper	7	8	2
paper	9	13	2
author	10	12	3
paper	14	21	2
reviewer	15	17	3
author	18	20	3

(A) Nodes table

WordNo	*Value*	*Lev*
4	Tim	4
11	Sarah	4
16	Ahmad	4
19	Wang	4

(B) Values table

Figure 3.10 The mapping of the *(Beg,End)* labeled tree in Figure 3.2 into relational tables

To evaluate a query by using this mapping, the query is translated into an equivalent SQL query and pushed down to the SQL engine for evaluation. The translation algorithm is illustrated in Algorithm 3.1.

Algorithm 3.1 : Evaluate an XPath Query

```
// F(Q)    :  Is a function to evaluate an XPath query and translate it into
              SQL query to retrieve answers from a relational data repository.
// Input   :  (XPath Query)
// Output  :  (SQL   Query)

1  $S(Q) \rightarrow \{Eq\}$        // Scan query Q and identify the element set of the given query.
2  $T(Q) \rightarrow$ SP or MP  // Identify type of query (Single path or Multiple paths).
3  If $\{Eq\}$ does not exists in $\{Es\}$ : // $\{Es\}$ is the element set of the Summary table.
      Then  :  abort and exit.
      Else  :  continue.
4  If $\{Eq\}$ query structure does not match $\{Es\}$ summary structure :
      Then  :  abort and exit.
      Else  :  continue
5  If $T(Q)$ == SP :      // If type of query is single path.
   5.1  $M(Q) \rightarrow \{(di,pi)\}$  // mapping of Q in Summary, identify leaf nodes IDs in Summary
   5.2  For each value in $\{(di,pi)\}$
            select  value
            from    value table
            where   Lev   = $di$   and
                    PerLv = $pi$
6  Else if $T(Q)$ == MP :// If type of query is multiple paths.
   6.1  $M(Q) \rightarrow \{(di,pi)\}$ //mapping of Q in Summary, identify leaf nodes IDs of all branching nodes .
        $M(Q) \rightarrow (Lb)$      // mapping of Q in Summary, identify the branching node level
   6.2  For each node in the first branch $\{d_1,p_1\}$ set:
         Begin
             Identify $s_1$ value at the branching node $(Lb)$
             For each node in $\{d_{1+i}, p_{1+i} : 1 \leq i < number\ of\ branches\}$
             Begin
                Find $s_{1+i}$ value
                if $s_1 == s_{1+i}$ then:
                     Select  value
                     From    value table
                     Where   Lev    = $d_{1+i}$  and
                             PerLv  = $p_{1+i}$
             End
         End
```

Step 1 of Algorithm 3.1 identifies the element set of a given query. The second step identifies the type of the given query, if it is single path or multiple paths query. Step 3 verifies that the element set of the given query exists in the element set of the Summary. Step 4 confirms that the structure of the given query exists in the Summary, as the Summary reflects the structure of the source data. If the tests performed in steps 3 and 4 succeed and the type of query is single path query, we evaluate the query as outlined in step 5. If the tests performed in steps 3 and 4 succeed and the type of query is multiple paths query, we evaluate the query as outlined in step 6. Step 5 evaluates single path query by retrieving the matching leaf nods of the query by using SQL statements. Finally, step 6 evaluates multiple path query by joining the first branch nodes with the matching nodes of the other branches of the given query by using the ids of the branching nodes. Then retrieve the related nodes from the database repository.

In the following, we show two examples that illustrate how this labeling scheme can be used to evaluate XML queries by using both regular and binary mapping. To evaluate the XPath Query 3.1 below over the mapped data in Figure 3.10, which is mapped by using regular mapping, the query is translated into SQL statement, and evaluated by the SQL engine. Our approach is flexible and can be used to translate an XPath query into several different sets of SQL statements. One of the possible translations is illustrated in SQL Query 3.2, as shown below. A survey on XML-to-SQL query translation can be found in Krishnamurthy et al. [75].

XPath Query 3.1 : */Bib/paper/author*

SQL Query 3.2 :

```
Select  Values.Value
From    Nodes Bib, Nodes paper,
        Nodes author, Values
Where   Bib.Tag     =  'Bib'          and
        paper.Tag   =  'paper'        and
        author.Tag  =  'author'       and
        Bib.Beg     <  paper.Beg      and
        Bib.End     >  paper.Beg      and
        paper.Beg   <  author.Beg     and
        paper.End   >  author.End     and
        author.Beg  <  Values.WordNo  and
        author.End  >  Values.WordNo  and
        Values.Lev  =  author.Lev+1   and
        Bib.Lev     =  1              and
        paper.Lev   =  Bib.Lev+1      and
        author.Lev  =  paper.Lev+1
```

Another possible mapping of *(Beg,End)* interval labeled nodes to relational tables is called the *binary mapping*. In this mapping, the *Nodes* table is horizontally partitioned based on the *Tag* name. That is, there is a dedicated table for each set of nodes with the same tag name. Evaluating the XPath Query 3.1 above over a *(Beg,End)* binary mapped data can be carried out by a translated SQL query that is similar to the SQL Query 3.2 above with the exception of the first 3 lines in the `Where` part. Here, the node selection part is eliminated, and the nodes are called in the `From` part from the designated tables.

An LLS labeled data-tree can also be mapped using the basic and binary mapping approaches. To evaluate XPath Query 3.1 over an LLS labeled data-tree that is mapped into relational tables using the basic mapping approach as shown in Figure 3.6, the query could be translated into the SQL Query 3.3 as shown below. The evaluation of XPath Query 3.1 over an LLS labeled data-tree that is mapped into relational tables using the binary mapping approach can be carried out in a similar way to the *(Beg,End)* binary mapped data evaluation.

SQL Query 3.3 :

```
Select  Values.Value
From    Summary Bib, Summary paper,
        Summary author, Values
Where   paper.Parent  = Bib.PerLv     and
        author.Parent = paper.PerLv   and
        values.Lev    = author.Lev    and
        values.PerLv  = author.PerLv  and
        Bib.Lev       = 1             and
        paper.Lev     = 2             and
        author.Lev    = 3
```

It is worth mentioning here that the binary mapping of LLS labeled data are finer than the binary mapping of *(Beg,End)* data. The LLS mapped data are partitioned horizontally based on the root path, not on the tag name, as in the

(Beg,End) labeled data. According to *(Beg,End)* labeled data binary mapping, it is possible for two nodes that belong to different root paths to be partitioned together in the same table if they have the same tag name. In contrast, this is impossible with the LLS labeled data. This explains the superiority of our LLS approach performance over the *(Beg,End)* approach performance as we shall see in the next section.

3.3 Prototype Implementation

The validity of our labeling scheme is illustrated by experiments conducted on a proof-of-concept prototype of the LLS that we implemented in our lab. All experiments were performed on a 3 GHz Intel® Pentium 4 PC running Windows® XP operating system, with 1.5 GB of RAM. The goals of the experiments are to evaluate the performance of the LLS labeling scheme in comparison to interval labeling schemes using the basic mapping and the binary mapping. Evaluation of the test queries against the datasets that are labeled with *(Beg,End)* labels is implemented by using Multiple Predicate MerGe JoiN (MPMGJN) as proposed by Zhang et al. [143]. We compare the basic and binary LLS labeling schemes with the basic and binary *(Beg,End)* labeling schemes, respectively. We also compare the basic mapping index structures against their respective binary mappings for both *(Beg,End)* and LLS node indexes. This allows us to observe the impact of horizontally partitioning the *Nodes* table into several tables based on the tag name in the *(Beg,End) Nodes* table, and the *Values* table into several tables based on the root path of the nodes in the LLS *Values* table.

We use IBM's DB2® V9.5 [64] database management system to store the data for the four schemes. We evaluate each method against two test sets (see Section 3.3.2). We measure the performance using the average runtime of a query with each method.

3.3.1 Query Engine Prototype to Test LLS Labeling Scheme

We implement the LLS as part of a query engine prototype shown in Figure 3.11. We use Java 1.6 to develop the prototype, which consists of 3 components:

1- The *scan* module, which scans an XML database according to a given configuration and then passes the contents of the scanned documents to the builder. The flowchart of the scanner is given in Appendix B.

2- The *builder* module collects the XML data from the scanner module. The data contains information about the type of the data (e.g. element, attribute, value, special characters, etc.) and the values of the data (element value, attributes value). The builder uses the LLS labeling scheme to label elements, attributes, and values then follows given configuration instructions to map the XML database into DB2® V9.5 [64] relational tables.

3- The *query engine* takes a query from a user and returns the solution to that query. The query engine is a primitive lightweight query processor. Basically, it translates an XPath query into an equivalent SQL query and passes it into the SQL engine for evaluation. It uses the data in the DB2® V9.5 backend to evaluate queries and return the answers.

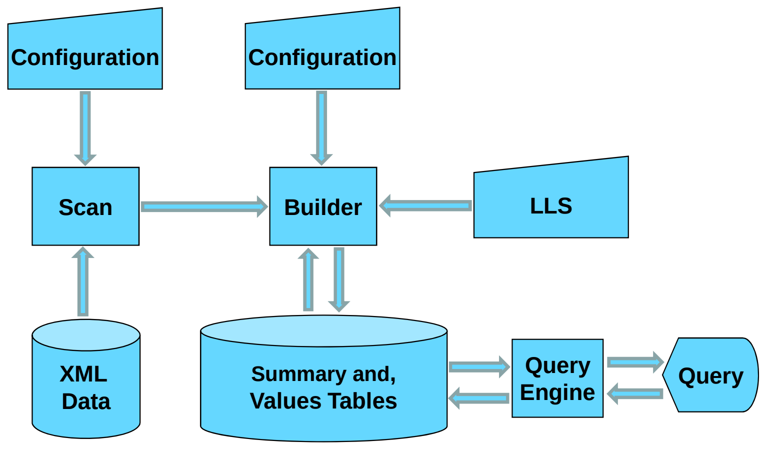


Figure 3.11 Layout of the query engine prototype to test the LLS

3.3.2 The Datasets and Queries

We execute our experiments using two datasets: the DBLP Computer Science Bibliography [120] dataset and the XMark [111] dataset. The DBLP dataset is record-oriented consisting of short data items such as name, title, date, etc. The XMark dataset, in contrast, consists of large data elements, many exceeding 7,000 characters. Statistics for the two datasets are summarized in Table 3.2. For more information on these datasets please see Appendix A.

Table 3.2 Statistics of DBLP and XMark datasets

Test Dataset	*Size*	*No of Elements in the Summary*	*No of Levels*	*Total Number of Elements*	*Max Cardinality*	*Avg. Cardinality*
DBLP	20 MB	71	5	582,033	109,595	8,197
XMark	15 MB	251	11	185,225	6,183	737

XPath (XML Path Language) [33] is a flexible query language that has been proposed to access XML data. An XML query may consist of either a single path or multiple paths (twig path). Both single path and twig path queries can be recursive (i.e. support ancestor-descendent "// " relationships) or non-recursive. Based on these criteria, we consider four types of XML queries :

Type 1 (T1): Single path non-recursive queries.
Type 2 (T2): Single path recursive queries.
Type 3 (T3): Twig path non-recursive queries.
Type 4 (T4): Twig path recursive queries.

Since most XML queries fall into these four types of queries, we use them in our experimental evaluation, and we run them against the two datasets. For each type of query, we use 4 example queries as shown in Figure 3.12. These queries are chosen to cover different combinations of query path lengths, cardinality of elements, and the number of returned tuples, which is affected by the selectivity. Figure 3.12 contain lists of the 4 types of queries, as specified by (T1,T2,T3, and T4). For more information on these queries please see Appendix A

T1-Q1 : */dblp/inproceedings/cdrom*
T1-Q2 : */dblp/inproceedings/cite/label*
T1-Q3 : */dblp/inproceedings/booktitle*
T1-Q4 : */dblp/book/series/href*
T2-Q1 : */dblp//author*
T2-Q2 : *//series/href*
T2-Q3 : *//book//label*
T2-Q4 : *//href*
T3-Q1 : */dblp/incollection[/year='2000']/booktitle*
T3-Q2 : */dblp/proceedings[/booktitle='ACCV']/isbn*
T3-Q3 : */dblp/inproceedings[/author='Adele E. Howe']/title*
T3-Q4 : */dblp/proceedings[/isbn='0-7695-1991-1']/title*
T4-Q1 : *//inproceedings[/mdate='2002-08-04']/title*
T4-Q2 : *//proceedings[/booktitle='ACNS']/isbn*
T4-Q3 : *//incollection[/booktitle='Temporal Databases']/year*
T4-Q4 : *//incollection[/author='Jurgen Annevelink']/title*

(A) For DBLP database

T1-Q1 : */site/regions/africa/item/id*
T1-Q2 : */site/open_auctions/open_auction/bidder/personref/person*
T1-Q3 : */site/open_auctions/open_auction/seller/person*
T1-Q4 : */site/catgraph/edge/from*
T2-Q1 : *//id*
T2-Q2 : *//africa//category*
T2-Q3 : *//regions//item//text*
T2-Q4 : *//open_auctions//text*
T3-Q1 : */site/regions/africa/item[/location='United States']/payment*
T3-Q2 : */site/regions/africa/item[/id='item0'] /location*
T3-Q3 : */site/catgraph/edge[/from='category0']/to*
T3-Q4 : */site/people/person[/name='Kaj Carey']/phone*
T4-Q1 : *//africa/item[/quantity='1']/name*
T4-Q2 : *//open_auction[/reserve='3199.90']/initial*
T4-Q3 : *//closed_auction[/type='Regular']/price*
T4-Q4 : *//regions//item[/quantity='2']/name*

(B) For XMark database

Figure 3.12 Representative queries for 4 types of queries

3.3.3 Performance Evaluation

We execute each query ten times against its respective dataset and take the average of the execution time of the 10 readings. The average of each type of the 4 types of queries is shown in our analysis. The time to translate the XPath queries to SQL queries is not included and only the execution times of the queries are recorded, which reflect the impact of the labeling scheme used.

Figures 3.13 and 3.14 shows a comparison between the *(Beg,End)* and LLS approaches performance against the DBLP test cases using basic and binary mappings. Figures 3.15 and 3.16 shows a comparison between the *(Beg,End)* and LLS approaches performance against the XMark test cases using basic and binary mappings. The results include the average runtime of the test cases. Note that log_{10} scale is used to measure the execution time. We notice from Figures (3.13-

3.16) that the LLS outperforms the *(Beg,End)* using both basic and binary mappings for the two datasets. The reason behind this is that the *(Beg,End)* approach, whether it uses basic or binary mapping, has to deal with the overhead that is caused by their partitioning techniques, where the partitioning techniques are based on the tag name, even though the partitioned nodes may belong to different paths. In contrast, LLS labeling lends itself to a more specific partitioning that is based on the root path partitioning, and hence only related nodes are examined as specified by a given query.

The experimental results show that in the case of the DBLP dataset, the improvement of our approach for basic and binary mappings is higher than that of the XMark dataset. This is because the number of elements in DBLP is greater than that of XMark dataset, as shown in Table 3.2.

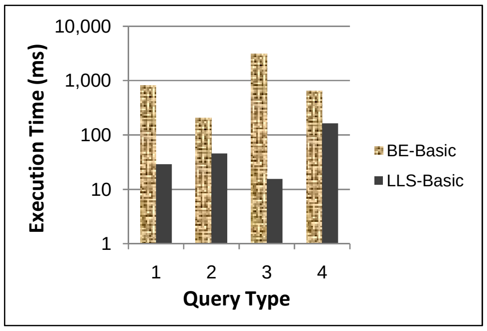


Figure 3.13 DBLP test cases result for the *(Beg,End)* and LLS using basic mappings

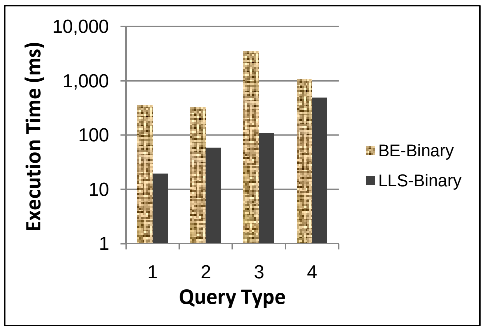


Figure 3.14 DBLP test cases result for the *(Beg,End)* and LLS using binary mappings

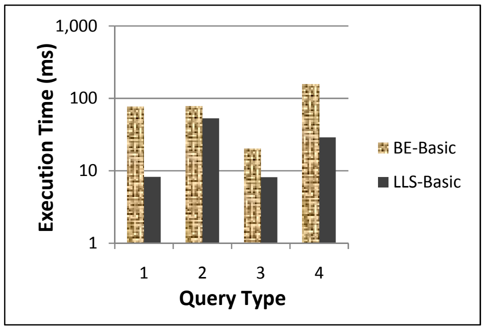


Figure 3.15 XMark test cases result for the *(Beg,End)* and LLS using basic mappings

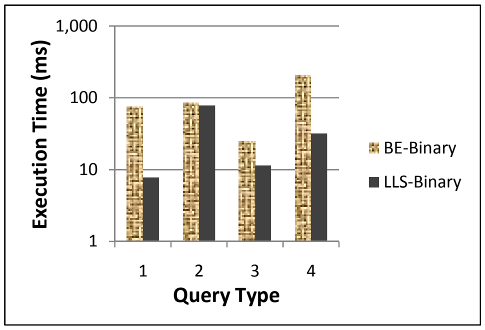


Figure 3.16 XMark test cases result for the *(Beg,End)* and LLS using binary mappings

Figures 3.17 and 3.18 show the performance comparison between the basic and binary mappings of the *(Beg,End)* and the LLS against the DBLP test cases. Figures 3.19 and 3.20 shows the performance comparison between the basic and the binary mappings of *(Beg,End)* and LLS against the XMark test cases. The results include the average runtime of the test cases. We notice from Figures (3.17-3.20) that the binary mapping performance, in general, is either approximately equal to or worse than that of the basic mapping. We believe that evaluating a query with *n* elements over a binary mapped data involves *n* tables and so a minimum of *n* disk accesses. While evaluating the same query over a basic mapped data would require accessing less disk accesses if the data happens to be clustered physically on the same pages.

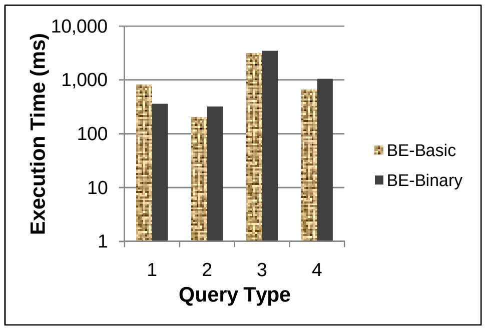


Figure 3.17 DBLP test cases result for the basic and binary mappings using *(Beg,End)*

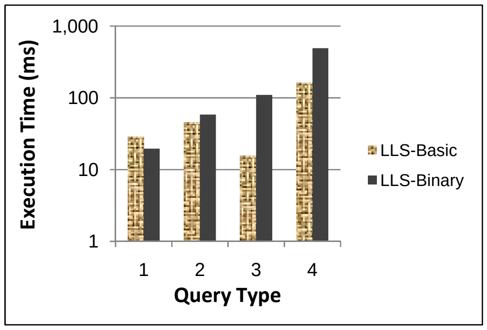


Figure 3.18 DBLP test cases result for the basic and binary mappings using LLS

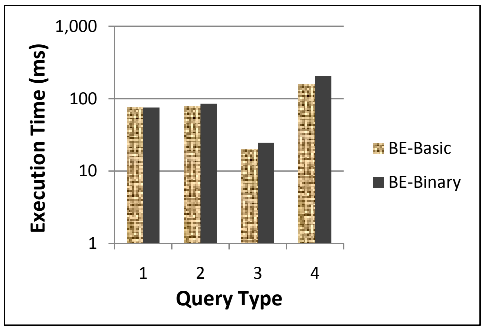


Figure 3.19 XMark test cases result for the basic and binary mappings using *(Beg,End)*

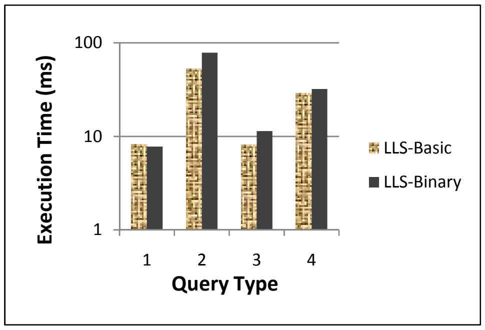


Figure 3.20 XMark test cases result for the basic and binary mappings using LLS

We believe that the performance gain over the LLS labeled data, as noted above, is due to the fact that LLS labeling scheme has the good characteristics of *(Beg,End)* labeling scheme, and on top of that , with LLS labeling we can infer the related nodes labels with the help of the summary. This feature helps to pinpoint the nodes of interest when evaluating a query instead of examining a wider range of nodes to find a match in query evaluation process as is the case in the *(Beg,End)* labeled data.

3.4 Summary

Unlike the two widely used labeling schemes – interval [63] [78] [116] [143] and prefix [48] [80] [99] [119] labeling schemes – our LLS labeling scheme contains the advantages of both approaches, while avoiding their disadvantages. The LLS is based on numerical values, has fixed-size labels regardless of the depth of a node, requires a fixed time to compute a relation between two nodes, can be used to infer the parent-child and ancestor-descendent nodes from their labels, and requires modest amount of relabeling upon insertion. Index structures that are based on the LLS are precise and complete.

The LLS is based on the level of nodes in XML data-trees. The information of the levels at which nodes are located can be used by the query processor optimizer to deduce nodes that may contribute a valid answer for a given query. The LLS also uses the summary of XML data to label the nodes in the data, so nodes that are related to a node of interest can be identified easily.

Much research has been done to propose a persistent labeling scheme for dynamic XML data to avoid the relabeling cost [23] [34] [77] [99]. Cohen et al. [34] established that any persistent labeling scheme requires $\Omega(N)$ bits per label in the absence of any clues about the data, where N is the size of the data. Such long labels, however, require high storage in addition to being more expensive to process than the shorter ones. In contrast, our labeling scheme, which is tightly coupled with the summary, requires a fixed label size to cover dynamic data.

The update cost of LLS, in the worst case, requires relabeling fewer nodes than that of the interval labeling scheme [93]. The cost of updating nodes using the LLS is also cheaper than that of updating prefix indexes in many cases.

Previous approaches that use a universal labeling scheme across a complete document (e.g. *(Beg,End)* approach) result in large labels for large documents. In contrast, in our approach we split the labels into groups of smaller numbers that require less memory and are easier to maintain and process than large labels.

We showed in a set of experiments the performance benefits of our proposed scheme compared with interval labeling schemes using regular and binary mappings to relational tables. The LLS works well for single path queries as well as for twig queries.

Chapter 4

LTIX: A Compact Level-based Tree to Index XML Databases

Indexing XML data is essential for XML query optimization. Most of the existing approaches that combine a labeling scheme with a graph index use labeling schemes that reflect the structure of the indexed data. In addition, the labeling rules do not depend on the combined graph indexes. By designing a labeling scheme that does not reflect the structure of XML data, since it is available in the accompanied graph index; and by aligning the data nodes' labels with the graph index nodes' labels, we can support the join process more efficiently. In this chapter, we propose a novel native XML index structure called LTIX (Level-based Tree Index for XML databases). This index structure is based on Level-based Labeling Scheme (LLS) that not only minimizes the number of joins and matches required to evaluate twig queries, if it is used with graph indexes, but also facilitates effective query optimization through early pruning of the space search. Experimental tests show the performance benefits of our proposed approach.

In this chapter we review XML structural indexes, formally define our index structure, and explain the LTIX system [95]. We conclude by presenting our experimental results and contributions.

4.1 XML Structural Indexes

In this section we discuss hybrid XML index structures, and we briefly review the weaknesses of XML structural indexes. We then introduce our LTIX approach, which we propose to overcome these shortcomings.

4.1.1 Hybrid XML Index Structures

Some researchers combine node indexes with graph indexes to expedite query processing and reduce the number of structural joins. For example, Kaushik et al. [69], Moro et al. [97], and Haw et al. [63] integrate the (*Beg,End*) interval node index with the DataGuide graph index. In these approaches element labels are assigned and then subsequently associated with their designated nodes in the graph index. In this case, the graph indexes, as well as the interval node indexes, hold the structural information of the data. We believe that it is sufficient for only one of them to hold the structure information in order for them to work well together. We can therefore plan a labeling scheme that is structure independent and link it with a graph index to provide the structural information. We implement this concept in our approach. Our approach is therefore similar to the approaches that integrate interval node indexes with DataGuide graph indexes [63] [69] [97] with the exception of the labeling scheme. We propose a novel index structure that is based on the LLS labeling scheme [94], which is shown to work efficiently with DataGuides.

4.1.2 Limitations of XML Structural Indexes

The main shortcoming of node indexes is the number of structural joins required to evaluate a query, which is equal to *n-1* where *n* is equal to the number of nodes in the query.

Graph indexing schemes [2] [37] [54] [56] [67] [90] consider paths as a whole, during query evaluation, instead of dealing with each node in the path separately. Consequently, the number of joins is reduced during query processing and hence query performance is improved.

Sequence indexes [104] [130] interpret the whole query as a structure-encoded sequences and search for a match in the structure-encoded sequence of an XML document. They suffer, however, from false positive and false negatives [93]. Refinement steps are added to the evaluation process of a query to overcome these problems.

An example of interval node indexes is shown in Figure 4.2. It is based on the (*Beg,End*) labeling scheme of the XML document in Figure 4.1.

```
<students>
    <student address="Kingston">
        <name>
            <fname>Tim</fname>
            <lname>Wang</lname>
        </name>
        <courses>
            <course>Art</course>
            <course>History</course>
        </courses>
    </student>
    <student address="Ottawa">
        <name>
            <fname>Sarah</fname>
            <lname>Ahmad</lname
        </name>
        <courses>
            <course>Math</course>
        </courses>
        <children>
            <child>
                <name>
                    <fname>Mike</fname>
                    <lname>Salem</lname
                </name>
            </child>
        </children>
    </student>
</students>
```

Figure 4.1 XML document

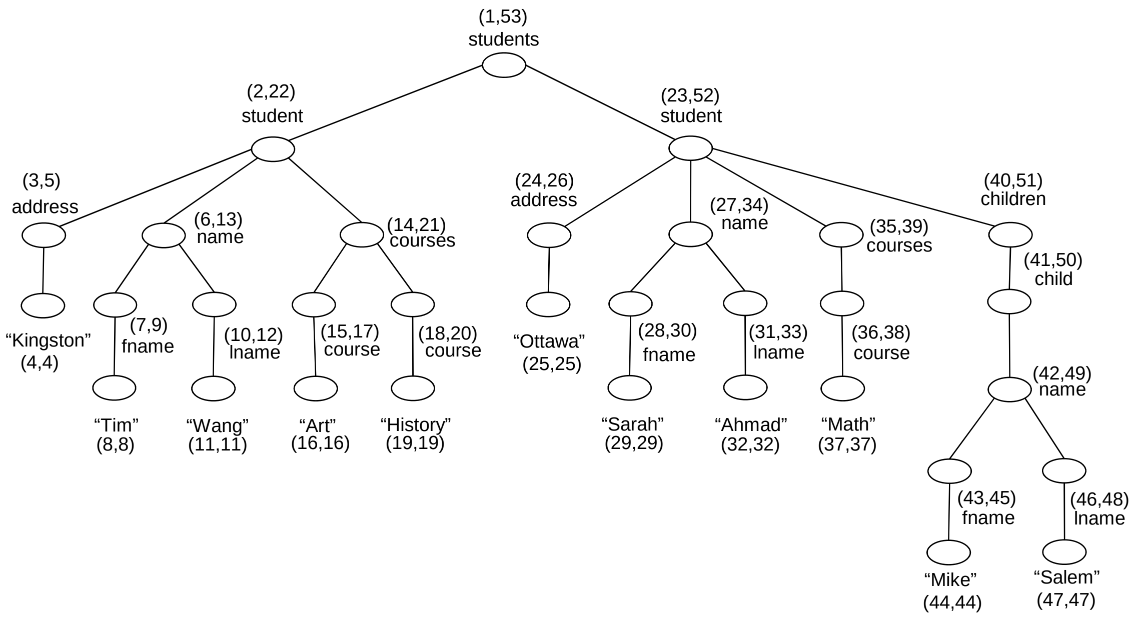


Figure 4.2 An interval labeled tree representation of the XML data in Figure 4.1

The labels are given according the sequential traversal of the document in Figure 4.1. In this type of node indexes, a relation between two elements is established if one element's interval contains the other element's interval.

Graph indexes partition element nodes in the source XML data-tree based on their path similarity. The DataGuide graph index in Figure 4.7 is an example graph index for the data-tree in Figure 4.2. The numbers inside the oval shaped nodes represent the labels of the graph index nodes. Unlike node indexes, which return the answers of XML queries at the granularity of individual instances of elements, graph indexes return the answers of XML queries at the granularity of the whole sets of instances of elements. Then a node index, such as the interval node index above is used to perform structural joins in a post-processing phase to arrive at the answers to a query. In the structural join operations, each element' instances in a set is compared with the other elements' instances in the other sets to find a match.

To evaluate a single path XML query a number of joins and comparisons are required if we use node indexes. To overcome this shortcoming node indexes can be integrated with graph indexes. To illustrate this consider evaluating Query 4.1 below over the data in Figure 4.2. Query 4.1 returns the first and the last names of the students in an XML document. The node-labeled tree representation for Query 4.1 is given in Figure 4.3.

Query 4.1: *//student/name[fname]/lname*

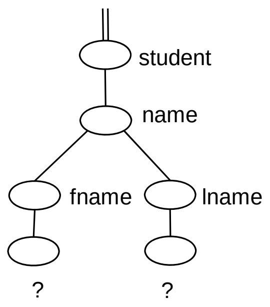


Figure 4.3 The node-labeled tree representation of Query 4.1

The instances of Query 4.1 elements in the XML data-tree in Figure 4.2 are saved in a node index structure similar to the one shown in Figure 4.4 as suggested by Zhang et al. [143], which is based on (*Beg,End*) labeling scheme.

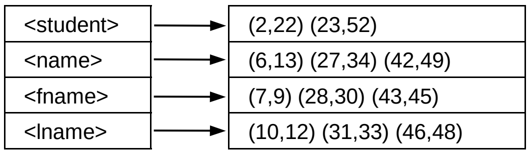


Figure 4.4 The (*Beg,End*) interval node index for instances of Query 4.1 elements in Figure 4.2

To evaluate Query 4.1 over the data in (*Beg,End*) interval node index in Figure 4.4, we need to implement 3 structural joins and 18 matches, in the worst case. This worst case is reached if we used the standard merge join algorithm to arrive at the final answer that contains the following tuples:

fname	**lname**
Tim	Wang
Sarah	Ahmad

Zhang et al.[143] propose the Multiple Predicate MerGe JoiN (MPMGJN) algorithm to reduce the number of joins. Much subsequent research has been done in this area to reduce the number of joins and comparisons [6] [18] [30] [76]. Discussion of these approaches is beyond the scope of this chapter.

To evaluate Query 4.1 above by using an integrated system such as the one shown in Figure 4.5, which integrates the DataGuide graph index (Figure 4.7) with the interval node index (Figure 4.4), we need to perform 2 join operations and 8 matches in the worst case. This worst case is reached if we used the standard merge join algorithm.

Level	*PerLv*	*Tag*	*Start*	*End*
2	11	student	2	22
2	11	student	23	52
3	21	name	6	13
3	21	name	27	34
4	11	fname	7	9
4	11	fname	28	30
4	21	lname	10	12
4	21	lname	31	33
5	11	name	42	49
6	11	fname	43	45
6	21	lname	46	48

Figure 4.5 Integration of the node index (Figure 4.4) with the graph index (Figure 4.7)

From the above discussion we notice that integrating interval node indexes with graph indexes dropped the number of joins, in our example, from 3 to 2 join operations, and the number of matching operations have been reduced from 18 to 8 matches. Our proposed LTIX approach, which integrates a special labeling scheme (LLS) with a DataGuide graph index, requires implementing only one join operation during which two matches are performed in the worst case, to evaluate Query 4.1 above. We will return to Query 4.1 example and explain how we can achieve this by using LTIX approach in Section (4.2.2) after we elaborate on our approach in the next section.

LLS labeling [94] scheme preserves the best traits of both interval labeling [63] [143] and prefix labeling schemes [99] [119] (See Chapter 2). Similar to interval labels, the size of LLS labels is fixed regardless of the data-tree depth, and hence requires modest storage space. Like interval labeling, integers are used to label nodes with LLS, which is more efficient for queries processing than the substring labels that are used in prefix labeling. Furthermore, a relation between two nodes can be identified with a single equality comparison operation with LLS, while with interval labeling, a relation is identified using two inequality comparison operations.

ORDPATH labels [99] are a variant of Dewey prefix labels [34] [99]. They do not need to be updated when new nodes are inserted, but they suffer from the shortcomings of prefix indexes. In the interval node index approach proposed by Zhang et al. [143], they suggest including the level of elements as a part of node labels. In contrast, our approach not only has the level of elements as part of the node labels, but we provide a graph index (absent from Zhang's approach), and the levels are also added to this graph index node labels.

4.2 Our LTIX Approach

In this section, we first introduce the XML data model used in LTIX, the graph index, and the mapping of XML data-tree into native XML graph index and data repository. We then trace two examples to demonstrate how LTIX is used to evaluate twig queries and to improve the efficiency of query evaluation process. We discuss LTIX path index construction at the end of the section.

4.2.1 XML Data and Path Index Models

We model an XML document as a directed graph $G=(R,V_R,V_L,E,tagg,labelg,T)$. The definition of this data model is given in Section 3.2.1. The LTIX data-tree graph representation *G* for the data in Figure 4.1 is illustrated in Figure 4.6.

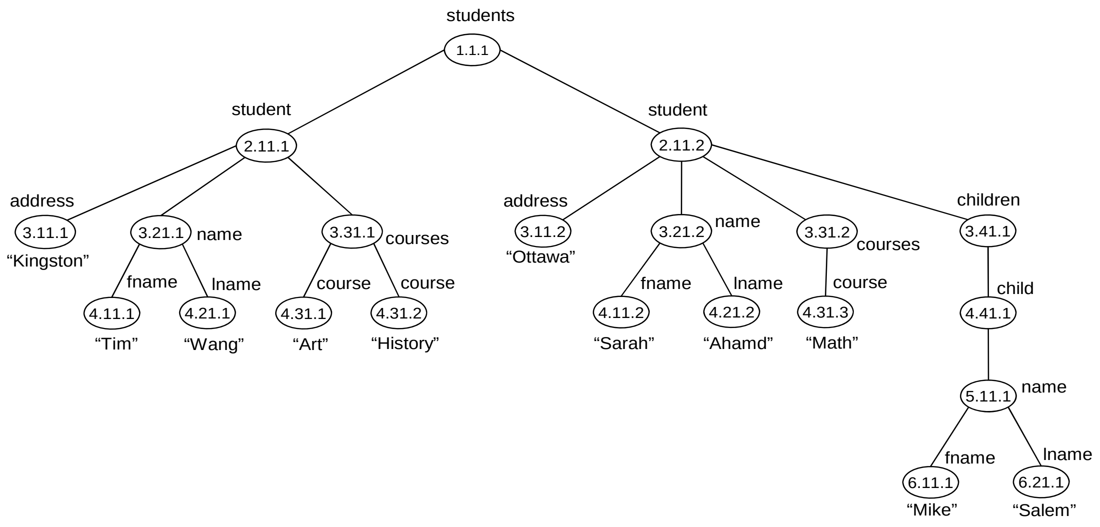


Figure 4.6 LTIX data-model of the data in Figure 4.1

In LTIX, an XML data-tree *G* can be summarized by a graph index *S* such that all node paths of *G* that share the same tag path *t* are represented by exactly one tag path *t* in *S*, and every tag path *t* of *S* is a tag path of at least one node path *n* of *G*. Basically, *G* nodes are partitioned into equivalence classes in *S* where the nodes of a class have the same root path.

We define a graph index as a directed graph $S=(R,O,M,tags,labels,C)$. Formal definition of the graph index *S* can be found in Section 3.2.1. Figure 4.7 contains an example of a graph index *S* of the XML data-tree *G* in Figure 4.6.

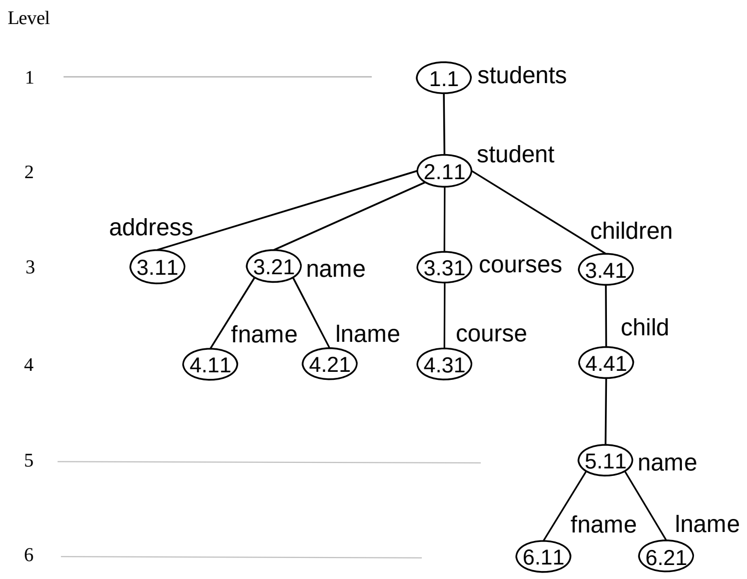


Figure 4.7 The graph index *S* of the XML data-tree *G* in Figures 4.2 and 4.6

The graph index *S* information of Figure 4.7 is mapped into table representation as shown by the *Graph Index,* and *Elements and Attributes Dictionary* tables in Figure 4.8 (B and A), and the data-tree *G* information of Figure 4.6 is mapped into table representation as shown by the *Value Index*, and *Elements and Attributes Index* tables in Figure 4.8 (C and D). We implement the G*raph Index* as a binary file; and the *Elements and Attributes Dictionary, Value Index,* and *Elements and Attributes Index* as B+-trees in our LTIX system. The key of each index is underlined in Figure 4.8.

Tag	Level	PerLv	Type
address	3	11	A
child	4	41	E
children	3	41	E
course	4	31	E
courses	3	31	E
fname	4	11	E
fname	6	11	E
lname	4	21	E
lname	6	21	E
name	3	21	E
name	5	11	E
student	2	11	E
students	1	1	E

(A) Elements and Attributes Dictionary

Level	PerLv	Parent
1	1	0
2	11	1
3	11	11
3	21	11
3	31	11
3	41	11
4	11	21
4	21	21
4	31	31
4	41	41
5	11	41
6	11	11
6	21	11

(B) Graph Index

Level	PerLv	No	Value	SerialPath
3	11	1	Kingston	1,1,1
3	11	2	Ottawa	1,2,2
4	11	1	Tim	1,1,1,1
4	11	2	Sarah	1,2,2,2
4	21	1	Wang	1,1,1,1
4	21	2	Ahmad	1,2,2,2
4	31	1	Art	1,1,1,1
4	31	2	History	1,1,1,2
4	31	3	Math	1,2,2,3
6	11	1	Mike	1,2,1,1,1,1
6	21	1	Salem	1,2,1,1,1,1

(C) Value Index

Level	PerLv	No	Parent
1	1	1	0
2	11	1	1
2	11	2	1
3	11	1	1
3	11	2	2
3	21	1	1
3	21	2	2
3	31	1	1
3	31	2	2
3	41	1	2
4	11	1	1
. . .	. . .	. . .	. . .
4	41	1	1
5	11	1	1
6	11	1	1
6	21	1	1

(D) Elements and Attributes Index

Figure 4.8 Data dictionary and indexes

Figure 4.8(A) contains a list of all elements and attributes in the graph index and is referred to as the *Elements and Attributes Dictionary*. The *Tag* field is the tag of the elements or attributes in the graph index, which is assigned through the *tags* function of *S*. The *Level* and *PerLv* columns represent the *d* and the *p* parts of the graph index nodes labels, respectively, as indicated in Figure 4.7. These labels are allocated through the *labels* function of *S*. The *Type* represents the type of node (e.g. element or attribute).

The *Parent* field in Figure 4.8(B) holds the *PerLv* labels of the parent nodes, which are the *p* values of the parent nodes. The *Level* value of the parent node is equal to the current node *Level* value minus one, so we do not need to list the parent node's level in the *Graph Index*. We assign a zero value for the parent of the root node since it does not have any parent. Tables A and B in Figure 4.8

could be combined, as in the Summary table of the LLS approach in Figure 3.6(A), but we prefer to keep them separate because they are used in different stages of queries evaluation process.

Figure 4.8(C) shows the *Value Index* table, which is populated with the values of attributes and elements of the XML data-tree in Figure 4.6. The *Level*, *PerLv*, and *No* values together form the labels of the leaf nodes <*d.p.s*>, as shown in the data-tree in Figure 4.6. These labels are allocated through the *labelg* function of G. The *Value* field contains the values of all leaf nodes, and *null* for empty elements. Note that the value labels (which consist of *Level*, *PerLv*, and *No*) are the same as the element or attribute labels to which they belong. Finally, the *SerialPath* field contains the serial paths *r* of each node in the tree. It represents a vector of the *No* values of the nodes that constitute a path from the root node to the designated node. It is used in structural joins to evaluate twig queries, as we shall see in the forthcoming example in Section 4.2.2.

All nodes in the XML tree are represented by the *Elements and Attributes Index* as shown in Figure 4.8(D). The *Elements and Attributes Index* can be extended to have the serial paths of all attributes and elements similar to the serial paths of values, but it is not necessary in our approach. Note that the *Parent* values in table (D) are different than the *Parent* values in table (B). In table (D) they stand for the *No* value of the parent node.

In order to achieve high performance of the LTIX index structure, and since an *s* value uniquely identifies a node among other nodes of the same class, we create the *Serial Graph Index* that is based on the concatenation of (*Level,PerLv,SerialPath*) of values. The serial graph index is used to facilitate the link between two arbitrary nodes in two different branches of a twig query as we shall see shortly.

4.2.2 Two Simple Examples

In this section we trace two examples. The first example shows how LTIX is used to minimize the number of join and comparison operations. The second example illustrates the ability of the LTIX approach to prune false positives early during the evaluation process.

Example 4.1: We evaluate Query 4.1 below, which was introduced in Section 4.1.2, over the mapped data in Figure 4.8. This query returns the first name and the last name of all students.

Query 4.1: *//student /name [fname] /lname*

Note that the branching occurs at the *name* node, which is the parent of the two leaf nodes, namely, *fname* and *lname*. We can see from the *Attributes and Elements Dictionary*, and the *Graph Index* that the *fname* and *lname* elements in Query 4.1 map to nodes <4.11> and <4.21> in *S*, respectively. First, we evaluate one side by probing the key fields of the *Value Index* for values whose labels start with ("4.11") and the two returned tuples are:

Level	*PerLv*	*No*	*Value*	*SerialPath*
4	11	1	Tim	1.1.1.1
4	11	2	Sarah	1.2.2.2

These tuples are joined with the *Value Index* to arrive at the final answer of the query. In order to do that, the information of these two tuples is used by the index structure as follows. We know that the *Serial Graph Index* is based on the concatenations of (*Level*,*PerLv*,*SerialPath*) columns. So the index structure probes the indexed columns in the *Serial Graph Index* for tuples that match (4,*21,LIKE 1.1.1%*), which is retrieved in one match. The *LIKE 1.1.1%* part retrieves all *SerialPaths* values that start with *1.1.1*. The search for a match to the second tuple is carried out in the same way by the search criterion (*4,21, LIKE 1.2.2%*), which retrieves the answer to this part in one match. The first three segments of the S*erialPath* (*"1.1.1"* and *"1.2.2"*) are used in the search criteria because the branching node is located at the third level. This means that the first three segments of the *SerialPath* of the two branches of the query are common and shared by the two branches.

Our approach, in contrast to the two approaches discussed previously in Section 4.1.2 – the (*Beg,End*) interval node index approach, and the approach that integrates the (*Beg,End*) interval node index with the DataGuide graph index – performs only one join during which two matches are performed to evaluate the query. In our approach, the leaf nodes of the two branches are matched directly with each other without using the branching node as a mediator to join them, as opposed to the previous approaches. Further, the information of tuples obtained from evaluating the first branch leaf node is used to retrieve the exact match in one comparison for each match by using an equality operator. The previous approaches, in contrast, require multiple comparisons to find a match, since there are two parameters involved in the searching criteria (*start*, and *end*), and both are involved in an inequality comparison (*less than "<"*, or *greater than">"*).

Note that the first two fields of the *Value Index* key and the *Serial Graph Index* key are the same (*Level*, *PerLv*). This fact increases the chances of a successful memory hit when the search criteria run against the *Serial Graph Index* are met by multiple tuples, and thus decreases the number of disk accesses. This clustering helps to explain the shortest response time achieved by LTIX system in comparison to the previous approaches as shown in our experiments in Section 4.3.

Example 4.2: The level of XML elements in graph indexes can be used to identify the elements' position within an XML tree structure, and can facilitate effective query optimization through early pruning of the space search. To demonstrate that, we evaluate Query 4.2 below over the mapped data in Figure 4.8. This query returns the values of *fname* elements that have a *child* element ancestor.

Query 4.2: *//child//fname*

Based on whether the used graph index carries the level information of the indexed elements or it does not, we have two scenarios to evaluate Query 4.2. First, if we assume that we do not have the level information in the graph index *S*, or we have it but we do not access it in an efficient way at an early stage of the evaluation process of a query, then we evaluate Query 4.2 as follows. We access the graph index and search for all *child* and *fname* elements. In this process node <4.41> of *child* element, plus nodes <4.11> and <6.11> of *fname* elements are retrieved and investigated. Node <4.11> is excluded as the structure index would indicate that it is not a valid choice.

The second scenario takes place if we assume that the levels of the graph index nodes are given and used at an early stage of the evaluation process. In this case, we match the extent of node <4.41> with only the extent of node <6.11>. The extent of node <4.11> would be excluded at an early stage since its level is equal to the level of <4.41> node, which contradicts the query specification. Our index structure includes the level at which a node is located as part of the node label in the graph index. Based on this fact, the evaluation algorithms of our LTIX approach detect invalid choices at an early step of the evaluation process and exclude them, thus improving the performance of query evaluation. To illustrate this, the *Elements and Attributes Dictionary* table in Figure 4.8(A) can be used in our approach to evaluate Query 4.2 as follows. If we follow a top-down evaluation plan, then we would use the *Elements and Attributes Dictionary* table to find the *child* element information first. The search will return the following tuple:

Tag	*Level*	*PerLv*	*Type*
child	4	41	E

This information is used as a predicate to search for *fname* elements that are located at a level greater than 4. This way, the node <4.11> in the graph index of *fname* element is excluded instantly without retrieving it. In contrast, other approaches will eventually exclude it, but after retrieving and testing it.

4.2.3 LTIX Path Index Construction

Graph indexes, in general, require a large amount of memory [56] [144]. Motivated by this fact, versions of graph indexes, called approximate indexes

[55] [67] [70] have been proposed to reduce the memory requirements. The memory reduction, however, comes at the expense of accuracy, and initial answers to a query are often subject to refinement steps to finalize the answers and remove false positives. Graph indexes, furthermore, are used heavily in XML query evaluation, especially for paths that have recursion. Finally, since the labeling scheme of our approach is based on the level of XML elements, our algorithms use graph indexes more often than other approaches to evaluate XML queries. Because of these facts, we discuss different alternatives to minimizing the size of the graph indexes without compromising their accuracy.

We propose two types of implementations for graph indexes. The first is called a *Matrix Index* and the second is called a *Flat Index*. Figures 4.10 and 4.11 are examples of the implementations of the first and the second types for the graph index in Figure 4.9, respectively. Figure 4.9 is a portion of the graph index in Figure 4.7. To simplify our examples we narrow the expansion gaps. The expansion gaps are reserved for inserting new nodes between the existing nodes while maintaining the order of the nodes.

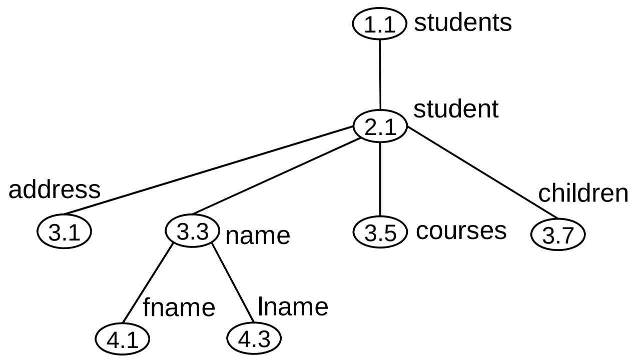


Figure 4.9 Fraction of the graph index in Figure 4.7

In the matrix index (Figure 4.10), sequential memory slots (numbered in the bottom- right corner) are dealt with as if it is a matrix that has two coordinates. The *X* and *Y* axes coordinates represent the width (*PerLv*) and the depth (*Level*) of the graph index. The first seven slots are reserved for the first level elements, the second seven for the second level elements, and so on. The width of the matrix is chosen in a way to cover the width of the tree, which is 7 in our example. Each slot contains the *Parent* value for its corresponding node in the graph index. The empty slots can be used for expansion. This index-probe operation is illustrated in Algorithm 4.1, which finds the parent node label for a given node. Note that "*V(U)*" in line 2 in Algorithm 4.1 represents the value that occupies the memory unit *U*. The matrix structure index does not have to have equal width and depth as in our example. The depth and the width of the matrix index may vary depending in the depth and the width of the graph index tree, and the formed structure would still maintain the uniformity of access, which is based on the chosen depth and width.

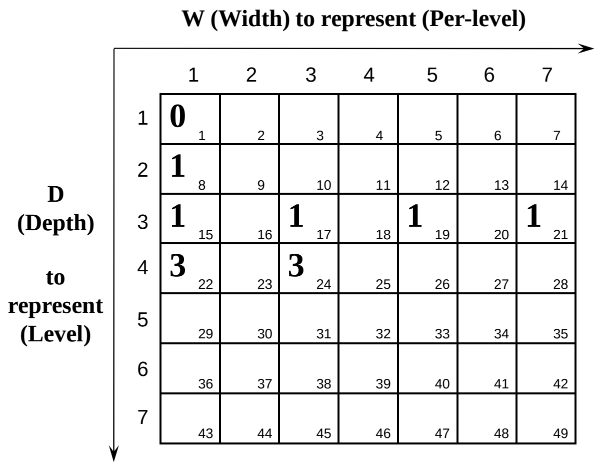


Figure 4.10 The Matrix Index structure that holds the *Parent* value of the graph index in Figure 4.9

Algorithm 4.1: Find the parent node of a given node using the Matrix Index

```
// F(di,pi) :  is a function to find the parent node label for a given node.
// Input    :  (di,Pi) is the label of a node where di and Pi are the Level and
               the PerLevel of the input node, respectively.
// Output   :  (do,Po) is the label of a node where do and Po are the Level
               and the PerLevel of the output(parent) node, respectively.
1  $do = di - 1$;                          // return the Level of the output node
2  $Po = V(\mathcal{U}) = V(((di-1)*W)+Pi)$   // return the PerLevel of the output node
```

Due to the increase in the number of nodes in the graph index as levels increase, and due to the fact that this index structure width has to be the same for all levels, the matrix would be sparse. Motivated by this fact, we propose the flat index (Figure 4.11) that divides the graph index into three parts. In the first part, we save the number of levels of the graph index (assume it is equal to *n*). The second part contains *n* storage units. These storage units are used to specify how many nodes there are in each level. For example, level 3 (in storage unit 4) has room for seven nodes. Finally, the third part contains the *PerLevel* of the parent node of all nodes in all levels, if they exist. Otherwise, *null* value is presented. This index-probe operation is illustrated in Algorithm 4.2.

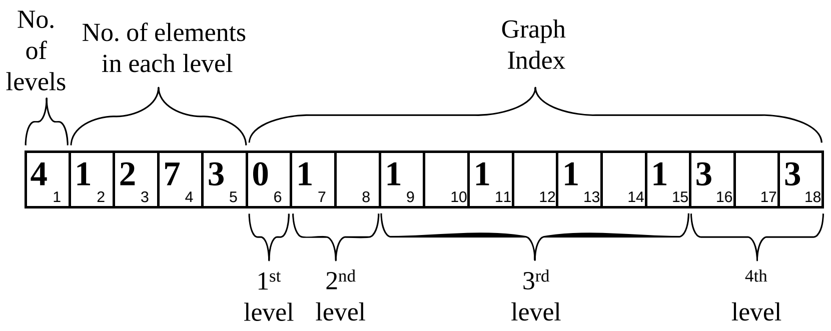


Figure 4.11 More efficient dynamic Flat Index structure for the graph index in Figure 4.9

Algorithm 4.2 : Find the parent node of a given node using the Flat Index

```
// F(d_i,p_i) :  is a function to find the parent node label for a given node.
// Input      :  (d_i,P_i) is the label of a node where d_i and P_i are the Level and the
                 PerLevel of the input node, respectively.
// Output     :  (d_o,P_o) is the label of a node where d_o and P_o are the Level and
                 the PerLevel of the output (parent) node, respectively.
1  Y <= V(1) ;                    // Assign the value in storage unit 1 to variable Y
2  Target = T = 0 ;               // initialize the value of target level
3  For k=2 to d_i                 // This loop is to find the address
4      { T=V(k)+T }               // of the level specified by d_i
5  d_o = d_i-1;                   // returns the value of d_o
6  P_o =V(u)=V(1+Y+T+P_i )        // returns the value of P_o
```

In order to evaluate a query *Q*, for example, */students//fname* against the mapped data of *G* in Figure 4.8, we first verify that the two elements of *Q* exist in *S* (the Elements and Attributes Dictionary, Figure 4.8(A)). If so, we get their labels, which consist of two sets of labels, {<1,1>} and {<4,11>,<6,11>} for *students* and *fname* elements, respectively. We then use Algorithm 4.3 to verify if a relationship exists between the instances of these two sets of elements before going any further in the query evaluation. The function *R*((*d1*,*p1*),(*d2*,*p2*)) in

Algorithm 4.3 is used to verify a child-parent or descendant-ancestor relationship between any two arbitrary nodes.

Algorithm 4.3: Confirm a relationship between two given nodes

```
// R((d1,p1),(d2,p2))  is a function to find if a relationship
                        exists between two arbitrary nodes.
// Input   :  (d1,p1) is the node in higher level and
              (d2,p2) is the node in lower level.
// Output  :  Boolean value: true if the relationship exists,
                 or false otherwise.
1   $n = d_2 - d_1$;
2   $d_i = d_2$;
3   $P_i = P_2$ ;
4   for $t = 1$ to $n$
5      {  $(d_o,p_o) = F(d_i,p_i)$; // The function of Algorithm 4.1 or Algorithm 4.2 is used
6         $d_i = d_o$;
7         $P_i = P_o$  }
8   if ($d_o == d_1$ and $p_o == p_1$)
9           then return  true;
10          else  return  false;
```

The size of the matrix and the flat indexes are dependent on the number of spare space available for insertion and the size of the tree. The more space we have, the more robust the graph index structure will be, but at the expense of size. There is a trade-off between the graph index size and its ability to adapt to insertion. Flat index structure, however, has more control over the index size.

We believe that these types of index structure representations are useful for XML databases. They transform the irregularity of XML databases into regular data that can be accessed uniformly. Moreover, the address of a node itself is used as part of the information to reconstruct the index tree, that is, we use the address as a representation for *Level* and *PerLevel* information instead of saving them inside the file, and hence save memory. The size of the flat index is equal to $O(n + v)$ were n is the number of nodes in the graph index structure and v is equal to the number of levels in the graph index plus one. In real life situations, where the nodes in the graph index can reach hundreds or thousands of nodes, v becomes negligible compared to n, and hence the size of the graph index is approximately $O(n)$ of nodes that a graph index can hold. Since our graph index is based on the DataGuide graph index, the size is relatively small for a regular data-tree, and grows linearly for irregular data-trees, but does not exceed the size of the source data in the worst case [54] [56].

Another alternative for building the graph index are B+-trees, which handle growth gracefully. The B+-trees structure however may require more accesses than that required by our approach to retrieve specific information. Because B+-trees accesses depend on the size of the tree that dictates the depth of the tree. In addition, B+-trees require huge space compared to that required by our approach. Our graph index structure is similar to a dynamic hash index to some extent.

4.3 Prototype Implementation

We validate the LTIX with an experimental prototype that we implemented in our lab using Java 1.6. All experiments were performed on a 3 GHz Pentium 4 PC running Windows XP operating system, with 1.5 GB of RAM. The goals of the experiments are to evaluate the performance of our LTIX approach that uses the LLS labeling scheme. We therefore compare three different indexing methods. First, we implement a basic interval node index with the MPMGJN algorithm proposed by Zhang et al. [143]. Second, we modify the MPMGJN algorithm to use the graph index we described above. This allows us to observe the impact of our graph index on performance. Third, we implement our LTIX method, which consists of the LLS labeling scheme and our graph index. We evaluate the LLS labeling scheme's effect in the LTIX system by comparing it with the extended version of Zhang's interval labeling scheme. We have two different labeling schemes integrated with the same graph index so performance differences should be due to the labeling schemes.

We use the Berkeley B+-tree to store the data for the three schemes, and we use a binary file to store the graph index. We evaluate each method against two test sets (see Section 4.3.1). We measure the performance using two platform-independent criteria, namely the number of comparisons performed to establish relations between two elements and the number of cases pruned by the method, as well as the average runtime of a query with each method. The size of the tables is not measured since they depend on the B+-tree implementation.

4.3.1 The Datasets and Queries

We execute our experiments using two datasets: the DBLP Computer Science Bibliography [120] dataset and the XMark [111] dataset with scale factor (0.1). Statistics for the two datasets are summarized in Table 4.1.

Table 4.1 Details of DBLP and XMark datasets

Testing Dataset	*Size*	*No of Elements in PathIndex*	*No of Levels*	*Total Number of Elements*	*Max Cardinality*	*Avg. Cardinality*
DBLP	20 MB	71	5	582,033	109,595	8,197
XMark	15 MB	251	11	185,225	6,183	737

In our experimental evaluation we use the four types of queries listed in the previous chapter. We run them against the DBLP and XMark datasets. For each type of query, we used 4 queries as shown in Figures 4.12(A and B). Figures 4.12 (A and B) contain lists of the 4 types of queries, as specified by (T1,T2, T3, and T4). Please note that the test queries used in this chapter are the same test queries used in Chapter 3, since they are comprehensive and versatile.

T1-Q1 : */dblp/inproceedings/cdrom*
T1-Q2 : */dblp/inproceedings/cite/label*
T1-Q3 : */dblp/inproceedings/booktitle*
T1-Q4 : */dblp/book/series/href*
T2-Q1 : */dblp//author*
T2-Q2 : *//series/href*
T2-Q3 : *//book//label*
T2-Q4 : *//href*
T3-Q1 : */dblp/incollection[/year='2000']/booktitle*
T3-Q2 : */dblp/proceedings[/booktitle='ACCV']/isbn*
T3-Q3 : */dblp/inproceedings[/author='Adele E. Howe']/title*
T3-Q4 : */dblp/proceedings[/isbn='0-7695-1991-1']/title*
T4-Q1 : *//inproceedings[/mdate='2002-08-04']/title*
T4-Q2 : *//proceedings[/booktitle='ACNS']/isbn*
T4-Q3 : *//incollection[/booktitle='Temporal Databases']/year*
T4-Q4 : *//incollection[/author='Jurgen Annevelink']/title*

(A) For DBLP database

T1-Q1 : */site/regions/africa/item/id*
T1-Q2 : */site/open_auctions/open_auction/bidder/personref/person*
T1-Q3 : */site/open_auctions/open_auction/seller/person*
T1-Q4 : */site/catgraph/edge/from*
T2-Q1 : *//id*
T2-Q2 : *//africa//category*
T2-Q3 : *//regions//item//text*
T2-Q4 : *//open_auctions//text*
T3-Q1 : */site/regions/africa/item[/location='United States']/payment*
T3-Q2 : */site/regions/africa/item[/id='item0'] /location*
T3-Q3 : */site/catgraph/edge[/from='category0']/to*
T3-Q4 : */site/people/person[/name='Kaj Carey']/phone*
T4-Q1 : *//africa/item[/quantity='1']/name*
T4-Q2 : *//open_auction[/reserve='3199.90']/initial*
T4-Q3 : *//closed_auction[/type='Regular']/price*
T4-Q4 : *//regions//item[/quantity='2']/name*

(B) For XMark database

Figure 4.12 Representative queries for 4 types of queries

4.3.2 Performance Evaluation

We execute each query ten times against its respective dataset and take the average of the 10 readings. The average of each type of the 4 types of queries is used in our analysis. Tables 4.2 (A and B) show the results of the testing of DBLP and XMark test cases, respectively. The results include the number of pruned cases, the average number of comparison operations, and the average running time to execute the queries of the test cases. The pruning is due to the use of the graph indexes and the information about the elements' levels. We notice that the number of pruned cases in DBLP dataset is less than those of XMark datasets. This is due to two factors. First, the number of levels is higher in the XMark dataset. Second, the number of repetitive element names (elements with the same name) is also higher in the XMark dataset. Since more elements are tested in the twig queries, we notice that the number of pruned cases for twig queries is more than those of single path queries for both datasets.

In both test cases, the number of row pairs compared drops to zero for both types of single path queries (T1 and T2) when the graph index is incorporated, and hence the performance of our approach is similar to that of the extended approach. The basic interval node indexes require significantly more comparisons than the extended interval node indexes and LTIX because, with the latter two indexes, the correct set of answers is identified by the graph index. During this process, the labels of the nodes (which consist of *Level* and *PerLevel* parts) that match the exact answer criteria are identified by using the graph index, then used to retrieve the answer from the data in the B+-tree index. The data is clustered in the B+-tree by these labels so the retrieval times are much smaller than those of the basic interval node indexes.

We see similar performance improvements for both types of twig queries (T3 and T4) in both test cases. The extended interval node indexes compare less row pairs than the basic interval node indexes (79%-90% less comparisons), and LTIXs compares 97.9%-99.9% less pairs than the extended interval node indexes. Similarly, extended interval node indexes perform 46%-78% faster than the basic interval node indexes, and our approach outperforms the extended interval node indexes by 89%-99.6%.

Table 4.2 Average pruned cases, comparisons, and runtime for 4 types of queries.

(A) Against DBLP dataset.

Query Type	*Average Pruned Cases*	*Avg. No of Comparisons*			*Avg. Runtime (msec)*		
		Basic Interval	*Extended Interval*	*LTIX*	*Basic Interval*	*Extended Interval*	*LTIX*
T1	5	105,684	0	0	22,108	6	6
T2	2	32,376	0	0	15,545	12	12
T3	13	322,491	67,966	17	39,503	8,543	32
T4	14	316,409	67,384	61	35,016	18,991	96

(B) Against XMark dataset.

Query Type	*Average Pruned Cases*	*Avg. No of Comparisons*			*Avg. Runtime (msec)*		
		Basic Interval	*Extended Interval*	*LTIX*	*Basic Interval*	*Extended Interval*	*LTIX*
T1	29	25,915	0	0	3,250	7	7
T2	69	7,003	0	0	1,426	48	48
T3	205	32,805	3,272	12	2,304	738	22
T4	140	81,022	7,902	168	5,241	1,732	191

The experimental results of the twig queries (T3 and T4) show that in the case of the DBLP dataset, our approach performs 99.5% - 99.6% faster than the extended approach, while in the case of XMark dataset our approach performs 89% - 97% faster than the extended approach. This is because the XMark dataset is text oriented where the size of data is very large and it exceeds 7,000 characters for many elements; while the DBLP dataset is record-oriented and the size of the data items is often short (e.g. name, title, date).

We believe that the performance gain of LTIX, as noted in Tables 4.2 (A and B), is achieved mainly by two factors in our index structure. First, the LTIX graph index is based on the levels of XML elements, which is used to prune out false positive cases early in the evaluation process. Second, multiple inequality comparisons are performed to find a match for a node using the basic and the extended node indexes, while LTIX only requires one equality comparison to find a match for a node.

4.4 Summary

Unlike the existing approaches that integrate a labeling scheme with a graph index, in which both reflect the structure of XML data, our approach relaxes the structure constrains from the labeling scheme of the integrated index structure. Alternatively, the used labeling scheme (LLS) not only facilitates effective query optimization through early pruning of the space search, but also is capable of supporting the join process more efficiently.

Graph indexes, in general, require a large amount of memory [56] [144]. Based on this fact, we developed several efficient implementations techniques for the graph indexes in LTIX to minimize the size of the graph indexes and increase its efficiency at the same time. Our indexing techniques are based on flattened indexes instead of B+-tree for two reasons. First, the B+-trees structure may require more number of accesses to retrieve specific information. Second, B+-trees require huge space to save them. In contrast, our graph index structures require less access to retrieve specific indexed information, and they require modest space.

Chapter 5

Relational Universal Index Structure for Evaluating XML Twig Queries

Numerous approaches to storing XML data in relational databases have been proposed in order to take advantage of the maturity of relational database management systems. Index structures have been developed with these approaches in order to speed-up XML query processing. However, these index structures typically either do not support efficient processing of twig queries or are huge in size. In this chapter we propose a novel index structure that is compact and effectively supports processing of XML twig queries. We use a light-weight native XML engine on top of an SQL engine to perform the optimization related to the structure of the XML data prior to shredding.

Experimental results show that our approach achieves lower response time than other similar approaches while using less space to store the XML data

In this chapter we discuss XML data mappings to relational data and we review the types of existing mapping approaches and their limitations. We then introduce our approach (UISX approach), explains the XML data and path summary models used to build UISX [96], explains how the proposed index structure optimizes the use of the space to store XML data, and illustrates how XPath queries are evaluated by using this index structure. Finally, we present an experimental evaluation of the UISX approach in comparison to existing approaches and summarize this chapter.

5.1 Mapping of XML Data to Relational Data

Due to its flexibility, XML is becoming the standard for exchanging data over the World Wide Web. XML databases can be stored and queried by using either native XML repositories [26] [53] [98] [102] [112] [136] [141] [142] [146] or relational database management systems [27] [49] [61] [101] [107] [114] [115] [118] [119] [134] [140] [143]. Native approaches for storing and querying XML data are still relatively new. On the other hand, RDBMSs are well founded, tuned, and standardized by several decades of work. RDBMSs are also known for their strength in data storage and manipulation, query processing and optimization, concurrency control, recovery, and security. Finally, huge volumes of data are already stored in relational database management systems. Motivated by these facts, researchers and vendors (such as IBM®, Oracle®, Sybase®, and Microsoft®) are working on ways to improve the capabilities of RDBMS to store and retrieve XML [25] [43] [44] [49] [61] [101] [114] [115] [118] [119] [140] [143].

5.1.1 Types of Mappings

Many research projects have proposed mapping XML data to RDBMSs. These proposals can be divided into two groups: mappings that are based on the schemas of XML data [126], which are referred to as *structure-mappings*; and mappings that are not based on XML schemas, which are referred to as *model-mappings* [118]. In structure-mapping, XML data is mapped to different relational schemas depending on the existing XML schemas. In model-mapping [91], the XML data is mapped to the same relational schema regardless of the structure of the mapped data, whether an XML schema exists or not. Shanmugasundaram et al. [115] and Florescu et al. [49] proposed two of the early approaches for mapping XML data. The first approach is based on structure-mapping, and the latter is based on model-mapping. Since our approach is based on model-mapping, we are going to concentrate on model-mapping approaches.

5.1.2 Problems with Model-Mapping Approaches

There are three types of model-mapping approaches: edge, node, and path approaches. The edge model-mapping approach proposed by Florescu and Kossmann [49] is based on the edge-labeled data model. It maps all edges in an XML data-tree into a single relational table that has the scheme (*Source*,*Target*,*Tag*,*Flag*,*Value*). Each edge represents an element that has a *Source* and *Target* identification. An XPath query is evaluated by matching the *Target* id of one element (edge) with the *Source* id of the following element in the path of a query starting from one end and finishing at the other end. The *Flag* represents the type of the node (e.g. int, string). The edge approach requires a minimum of *n-1* join operations to evaluate a query with *n* elements for both single path and twig path queries. In addition, it does not efficiently evaluate queries with the ancestor-descendent "//" axis.

Zhang et al. [143] proposed a model-mapping approach based on the node-labeled data model. They use intervals to label the nodes and map XML tree elements to a relational table that has the scheme (*Beg,End,Tag,Level,Value*). Two elements can be joined together if the interval (*Beg,End*) of one element contains the other element interval. Unlike the edge approach, node model-mapping can efficiently evaluate queries with the ancestor-descendent "// " axis, but it still requires *n-1* joins to evaluate a single path or a twig path query with *n* elements.

Yoshikawa et al. [140] proposed a model-mapping approach that is based on forward-paths of elements in an XML data-tree. A forward-path is a path that starts from an element in the higher part of an XML data-tree (e.g. root element) and ends at an element at the lower part (e.g. the mapped element). In this approach, elements are shredded into a relational table with the scheme (*Path,Beg,End,Value*). Each element is identified by its root-path (which is a forward-path). Single path queries are evaluated with one match. Twig queries, however, are evaluated by decomposing the twig into multiple single paths. Each path is evaluated separately and then joined together to obtain the final answer. The number of joins required to evaluate a twig query is usually equal to the number of branches in the query. The forward-paths approach reduces the number of joins required to evaluate a query, however, it may produce incorrect answers when recursion exists in XML data [56]. To overcome this problem Pal et al. [101] proposed a similar approach using reversed-paths instead of forward-paths. A reversed path is a path that starts from an element at a lower part in an XML data-tree and ends at an element in a higher part. The reversed-paths approach not only eliminates the possibility of producing false results, but also improves the performance of query evaluation. The reversed-paths approach has been used by IBM® System RX, Microsoft® SQL Server 2005, and Oracle® DB [56].

Chen et al. [27] use a reversed-path approach where each node in an XML data-tree is given a global id, and then shredded into relational tuples with the scheme (*HeadId*, *SchemaPath*, *LeafValue*, *IdList*). The *HeadId* is the id of the node at which a reversed-paths ends, *SchemaPath* represents the reversed-paths of XML data nodes, *LeafValue* represents the values of the leaf nodes in the path of the mapped elements, and *IdList* contains lists of the global ids of the nodes that constitute a path from the *HeadId* to the designated mapped nodes. Two index structures were proposed with the approach. The first is the ROOTPATHS index, which indexes only the prefixes of the root-to-leaf paths. The second is the DATAPATHS index, which indexes all subpaths of root-to-leaf paths, including the root-to-leaf paths. The key idea of this approach is to create an index for all branching nodes. To process a twig query, in the case of ROOTPATHS index, all branches are evaluated and the returned *IdLists* are then merged or hash-joined to arrive at the final solution. In the case of DATAPATHS index, a twig query is processed by evaluating the base branch (the branch that is evaluated first) to get the ids of the branching nodes which are available in the *IdList*. Then a search is carried out for the secondary paths that are rooted at the identified branching nodes and that have the exact reversed-path given in the query. The reversed-paths that are used to evaluate a twig query in DATAPATHS index start from the leaf node of the query and end at the branching nodes. The DATAPATHS index reduces access to the index to a single index lookup in order to find a match for fully specified, single path query without any recursion. Consequently, solving twig queries, which can be divided into multiple single path queries, requires a relatively small number of index lookups.

Chen's et al [27] index structure does not have a path summary table like our approach. Their approach, however, has a dictionary to encode schema paths by using special characters to designate elements and attributes instead of using the whole names. This dictionary has to be accessed at an early stage of an XML query evaluation process. Our approach, in contrast, uses the path summary table, which has approximately the same size as the dictionary table. The key idea of both approaches is to index all leaf nodes in relation to the branching nodes, and so minimize the number of index accesses required to evaluate a twig query.

5.1.3 Introduction to Our Approach (UISX)

Our approach falls under the model-mapping category and it is based on indexing branching nodes through which we can join XML data-tree nodes. The UISX is also based on a novel type of mapping, that is, structure summary mapping (summary mapping for short). In what follows, we introduce the Universal Index Structure for XML [96].

Elements in XML data are linked through a hierarchical structure. Any two elements are linked through their common ancestor. Therefore, indexing common ancestors can facilitate the evaluation of twig queries. For example, consider the XPath query 5.1.

Query 5.1: *//student [/ fname ='Sue' and lname ='Jones'] / program*

This query returns the program of the student *Sue Jones*. Its pattern can be represented as a node-labeled tree as shown in Figure 5.1. A single line represents a parent-child relation and a double line represents an ancestor-descendent relation.

Figure 5.2 contains an XML document, which is represented as the hierarchical node-labeled tree in Figure 5.3. The nodes' labels are given inside the nodes of Figure 5.3. Query 5.1 can be evaluated over the data in Figure 5.3 as follows. We first evaluate the branch with *fname*=*'Sue'*. This part returns the node <4.2.3> and the branching node <3.1.3>, assuming that all branching nodes for each node in the data-tree are recorded in the database. We call the branch that is evaluated first the *base branch*, and the branch(es) evaluated afterward the *secondary branch(es)*.

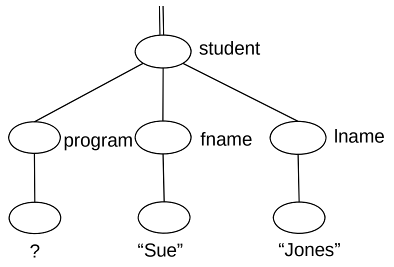


Figure 5.1 Query 5.1 hierarchical patterns

Now to evaluate the second branch *lname*=*'Jones'*, we have to search for the *lname* element that has a value "Jones" and whose parent node label is <3.1.3>. The only node that matches these criteria is node <4.3.3>. Note that the other two nodes that have the same last name "Jones," namely, nodes <4.3.1> and <4.3.2>, are excluded early in the search because their parent node is not <3.1.3>. Finally, we search for branch *program* relative to its parent <3.1.3>. So the value "CS" is returned as the final answer.

```
<course number="251">
   <name>XML</name>
   <students>
      <student>
         <program>Math</program>
         <fname>Omar</fname>
         <lname>Jones</lname>
      </student>
      <student>
         <program>Physics</program>
         <fname>Ayah</fname>
         <lname>Jones</lname>
      </student>
      <student>
         <program>CS</program>
         <fname>Sue</fname>
         <lname>Jones</lname>
      </student>
   </students>
   <instructor>Beth</instructor>
</course>
```

Figure 5.2 XML document

From the above example, we can see that twig queries can be evaluated by using knowledge of their branching nodes. We propose an approach that utilizes this idea to evaluate twig queries efficiently by building a Universal Index Structures for XML databases (UISX). This index structure guarantees to find a *complete* and *accurate* match for each node of any *arbitrary* base branch by executing a single index lookup. That is, all relevant matching tuples are retrieved without any false positives.

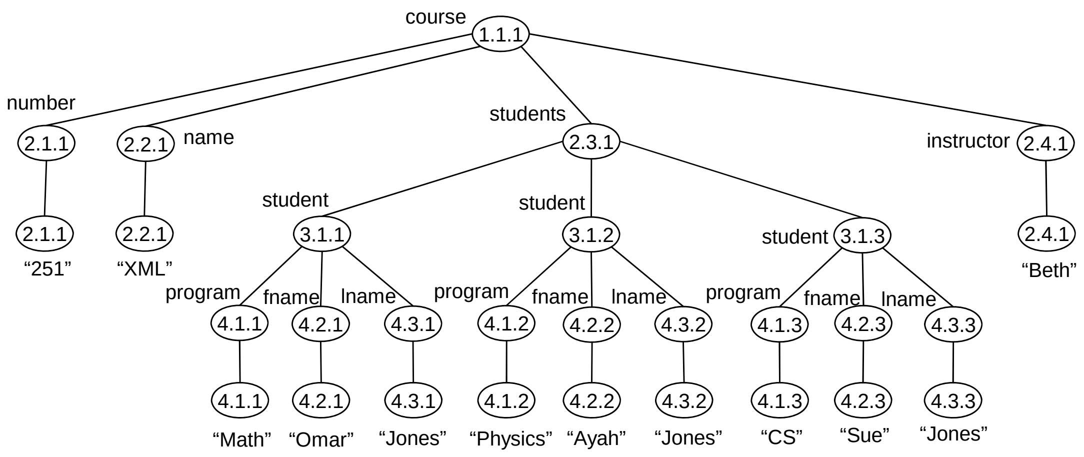


Figure 5.3 The data-tree representation of the XML document in Figure 5.2

Finding matching elements of a twig is a core operation in XML query processing [27]. Much research has been done to match elements at different branches of twig queries [27] [37] [54] [67] [88]. Generally, these approaches suffer from either being huge in size, or not being able to support twig queries as efficiently as they support single path queries. A good study of the trade-off between index space and evaluation efficiency is given by Chen et al. [27]. They implement two index structures: ROOTPATHS and DATAPATHS. ROOTPATHS has small size, but it is not as efficient as DATAPATHS, whose size is much larger. The reason behind the DATAPATHS superior performance is the fact that it indexes all possible subpaths of root-to-leaf paths, which are used to match any two arbitrary branches.

Our proposed approach has a compact size, yet, it supports efficient evaluations of twig queries. It uses a RDBMS to store and query XML documents. It also uses path summaries, which are based on DataGuides [54], to facilitate a query evaluation.

5.2 Our Approach: Universal Index Structure for XML Data

Based on the observation that branching nodes are the key element in solving twig queries, we propose the UISX approach [96] to efficiently match and join any two arbitrary nodes that share the same branching node. In this approach the base branch is evaluated first. Then, for each returned base branch node, the secondary branches are examined, and the matching nodes of each branch are located through their common ancestor node by using only one index lookup.

5.2.1 XML Data and Path Summary Models

In this section we describe our basic data model, and path summary that we used in developing the UISX system. Then we discuss the query language, the size optimization, and the query processor of the UISX.

We model XML documents as trees. An XML tree is a directed ordered graph $G=(R,V_R,V_L,E,tagg,labelg,T)$. Formal definitions of this data model can be found in Section 3.2.1. The data-tree representation *G* of the data in Figure 5.2 is illustrated in Figure 5.3.

In UISX, an XML data-tree *G* can be summarized by a path summary *S* such that every distinct path in the source data to appear only once in the path summary, and all the paths in the summary have to have at least one matching path in the original source data. Basically, *G* nodes are partitioned into equivalence classes in *S* where the nodes of a class have the same root path [54] [93].

We model path summary as a directed ordered tree $S=(R,O,M,tags,labels,C)$. Formal definition of the path summary *S* can be found in Section 3.2.1. Figure 5.4 contains an example of a path summary *S* of the XML data-tree *G* in Figure 5.3. Note that the leaf nodes' labels in *G* are represented by their parent nodes' labels.

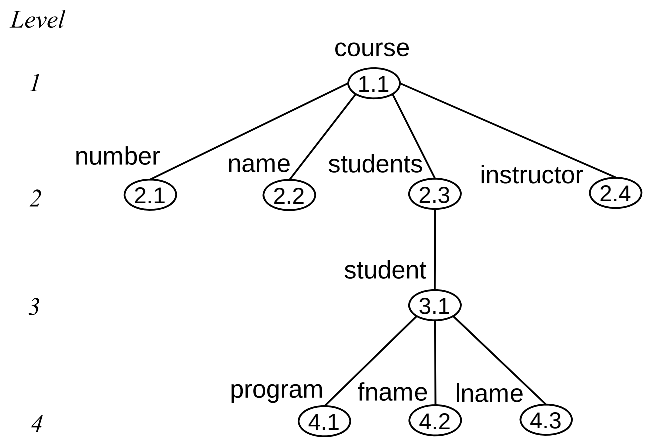


Figure 5.4 The path summary of the data in Figures 5.2 and 5.3

The path summary in Figure 5.4 is mapped to the relational table *PathSummary* as shown in Table 5.1, which is similar to the Summary table of LLS approach (Figure 3.6(A)). The leaf nodes data of Figure 5.3 is mapped to the relational table *LeafNodes* as shown in Table 5.2.

In Table 5.1, the *Tag* field contains the tags of the elements of the nodes in the summary, which is assigned through the *tags* function of *S*. The *Level* and *PerLv* fields represent the *d* and the *p* parts of the path summary nodes labels as indicated in Figure 5.4, respectively. These labels are allocated through the *labels* function of *S*. The *Parent* field holds the label of the parent nodes, which are the *p* values of the parent nodes. The *Level* value *d* of the parent node is equal to the current node *Level* value minus one, so we do not need to list the parent node level in the *PathSummary* table. Note that the *Parent* value of the root element is zero since it does not have a parent. The *Type* represents the type of node (e.g. element or attribute). The *Count* value *C* is the number of nodes in the original XML data that belong to the same summary group. It is used mainly to reconstruct the subtrees that are rooted at the internal nodes (see Section 5.2.3).

Table 5.1 The PathSummary table

Tag	*Level*	*PerLv*	*Parent*	*Type*	*Count*
course	1	1	0	E	1
number	2	1	1	A	1
name	2	2	1	E	1
stduents	2	3	1	E	1
instructor	2	4	1	E	1
student	3	1	3	E	3
program	4	1	1	E	3
fname	4	2	1	E	3
lname	4	3	1	E	3

Table 5.2 shows the *LeafNodes* table, which is populated with the data of all leaf nodes V_L in the XML tree. In this table, the *Level*, *PerLv*, and *No* values together form the label of the leaf nodes *d*, *p*, and *s*, respectively, as shown in the data-tree in Figure 5.3. These labels are allocated through the *labelg* function of *G*. The *Value* field contains the values of the node for V_T nodes, and *null* for V_E nodes. The *Lev1,…, Lev4* fields are explained below.

Table 5.2 The LeafNodes table

Level	***PerLv***	*No*	***Value***	*Lev 1*	*Lev 2*	*Lev 3*	*Lev4*
2	1	1	251	1	1	0	0
2	2	1	XML	1	1	0	0
2	4	1	Beth	1	1	0	0
4	1	1	Math	1	1	1	1
4	1	2	Physics	1	1	2	2
4	1	3	CS	1	1	3	3
4	2	1	Omar	1	1	1	1
4	2	2	Ayah	1	1	2	2
4	2	3	Sue	1	1	3	3
4	3	1	Jones	1	1	1	1
4	3	2	Jones	1	1	2	2
4	3	3	Jones	1	1	3	3

Branching Indices : In order to achieve high performance of the UISX index structure, and since an *s* value uniquely identifies a node among other nodes of the same class, we split the serial path parts that are separated by dots previously, and save each part in a different field (see Table 5.2). Each field is titled after the level of the *s* values it contains. That is, each field is titled *Lev*(*i*), where $i \in [1,\ldots,n]$, and *n* is the number of levels in *G*. Each field of these fields is used for indexing branching nodes located at the corresponding level. We define *H* as a set of *branching indices* that we create to index *si* values, where $H=\{H1,H2,\ldots,Hn\}$, $i \in [1,\ldots,n]$, and *n* is the number of levels in *G*. Each index for each level is based on the concatenation of (*si*,*d*,*p*) values (see Table 5.2). All *s* values of *V* nodes in *G* are covered by the *H* set.

The index structure of UISX has mainly three components: *path summary* table, *leaf nodes* table, and *branching indices*. The tables' key fields are underlined in Tables 5.1 and 5.2. The key field of the *PathSummary* table is (*Tag*), and the key fields of *LeafNodes* table are (*Level*,*PerLv*,*Value*). The two tables are related through the (*Level*,*PerLv*) fields. The branching indices are the *H* set of indexes, which are used to facilitate the link between two arbitrary nodes in a twig query. Our index structure covers nodes that belong to the same XML document; the extension to multiple documents can be implemented by adding the document id to the labels of *S* and *G* nodes.

5.2.2 X-Path Query Expressions

In what follows we introduce three definitions that are used in defining X-Path query expressions formally, as they are used in the UISX.

Definition 5.1. A query *Q* is covered by a summary S if and only if: (1) the nodes of *Q* exist in *S*, and (2) the *Q* nodes exist in *S* according the structure specified by *Q*.

For example, the query *"/instructor[/name='XML']/course"* is not covered by the path summary *S* in Figure 5.4. Although the first condition is met, but not the second. If we switch the positions of *course* and *instructor* tags *"/course[/name='XML']/instructor,"* then the mapping of *Q* nodes to *S* nodes succeeds and *S* covers *Q*.

Definition 5.2. To evaluate a twig query *Q* over a data graph *G* by using *S*, we say that the matching of an instance of one group with the instances of another group is *complete* if the returned nodes contain all the relevant nodes.

Definition 5.3. To evaluate a twig query *Q* over a data graph *G* by using *S*, we say that the matching of an instance of one group with the instances of another group is *precise* if the returned nodes do not contain any irrelevant node.

The pattern of single path query expressions can be represented as "$t_1.rel.t_2 \ldots rel.t_x$," where ($t_1,t_2,\ldots,t_x$) are tags of the query and *rel* represent the relation between the adjacent tags, and it may be a parent-child relation "/ " or ancestor-descendent relation "//." We refer to single path query expressions that have only the "/ " axis as simple single path queries, and to single path query expressions that have one or more "// " axes as single complex path queries. Both types are evaluated by finding the extension of *tx*, that is, $ext(t_x)$. In the relational tables in UISX, the mapped data are sorted by <*d.p*> keys, and hence one index look up is sufficient to evaluate these types of queries by probing the index for tuples that match <*y.x*>, where <*y.x*> is the label of t_x, and *d*=*y* and *p*=*x*. Twig queries patterns can be represented as:

$t_1.rel.t_2 \ldots rel.t_b[rel.t_1.rel.t_2 \ldots tf_1][rel.t_1.rel.t_2 \ldots tf_2] \ldots rel.t_1.rel.t_2 \ldots tf_i$

This twig pattern expression consists of multiple single path expressions. The expressions inside the square brackets and the expression that follows at the end are the branches of the twig. The branching element tags are denoted by *tb*. ($tf_1.tf_2 \ldots tf_i$) are the leaf elements' tags of the first branch, second branch, and i^{th} branch, respectively, where *i* is the number of branches in the twig. Given an XML data-tree *G* with a path summary *S*, in general, with UISX we evaluate a twig query *Q* against *G* in two steps. First, we map nodes of *Q* to nodes of *S*. If the mapping succeeds (i.e. *Q* is covered by *S*), we move to the next step in the evaluation process. In the second step we use only the extension of tags t_b and (tf_1, $tf_2, \ldots, tf_i$) to evaluate the query. Before we present an example, we need to introduce the following theorem.

Theorem 5.1. In UISX, one index lookup into a branching index *H* is sufficient to join a pre-defined node of one group of the leaf nodes with the complete and precise matching nodes in another group of leaf nodes of a twig query.

Proof. First consider the following twig query with two branches:

$\mathbf{Q}$ **:** $t_1.rel.t_2 \ldots rel.t_b[rel.t_1.rel.t_2 \ldots tf_1][rel.t_1.rel.t_2 \ldots tf_2]$

In this query, we assume that the level of the branching node t_b is L_b, and *Q* has two leafs: tf_1 and tf_2. The extension of tf_1 is a set of nodes Vf_1, that is, $ext(tf_1)=Vf_1=\{vf_{11},vf_{12},\ldots,vf_{1n}\}$, and similarly $ext(tf_2)=Vf_2=\{vf_{21},vf_{22},\ldots,vf_{2n}\}$, where *n* is the number of instances in each set. According to query *Q*, we want to prove that one index lookup into HL_b is sufficient to join a single node in Vf_1 node-set with all matching nodes in Vf_2 node-set.

From the data-graph G definition, the labels of Vf_j sets, where $j \in [1,2]$, consist of the three parts $<d.p.s>$. The first two parts (d and p) are the same for all nodes in each set. The third part s is the part that uniquely distinguishes each node among all nodes of the same class (or group). Each node in Vf_j sets has a serial path r (Definition 3.2), which consist of the s part of the labels of the nodes in the path from the root node to the designated node. Since s is unique for each instance of a class, then r can be used to uniquely identify the labels of all nodes in the serial path of a node. Assume that the value of s of the branching node t_b that is located at level L_b is s_x. The two branches' nodes that share t_b node in their serial paths are matched if the value of each serial path r at t_b node is equal to s_x value. This way, the matching process will return either an empty set if there is no match, or it will return the complete and precise matches since all nodes that share this common ancestor t_b node have their r values at t_b set to s_x. Consequently, there is no chance for any false positives to be retrieved. Since all s values of V nodes in G are covered by the H set of indexes (see branching indices), and HL_b index is based on s values of $Lev(L_b)$ field, then by using an index structure that contains HL_b, it would require only one index lookup to find a complete and precise match for any arbitrary node in one branch with one or more nodes in any other branch of a twig provided that they share a joining node. This matching process can be extended to evaluate multiple branches queries with n branches by evaluating two branches at a time until all branches are evaluated as illustrated in Algorithm 5.2 (to be discussed shortly). □

Example 5.1. Consider twig Query 5.2 over the data-tree *G* shown in Figure 5.3, which returns the list of students' first name and the programs in which they are enrolled.

Query 5.2: */course//student [/ program]/fname*

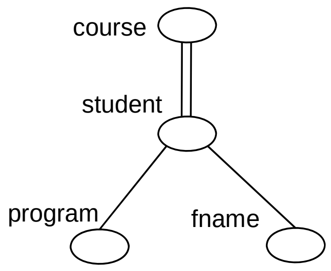


Figure 5.5 The node-labeled tree representation of Query 5.2.

This query node-labeled tree representation is shown in Figure 5.5. It is easy to see that *S* covers *Q* because the mapping of *Q* query over *S* path summary of *G* data-tree can be carried out successfully. In this case *student* node is the branching node t_b, *program* node is the first leaf node tf_1, and *fname* node is the second leaf node tf_2. These three *Q* nodes maps to *S* nodes <3.1>, <4.1>, and <4.2>, respectively. Note that L_b=3. We next retrieve *ext*(tf_1), the extension of tf_1, which returns the tuples:

Level	*PerLv*	*No*	*Value*	*Lev 1*	*Lev 2*	*Lev 3*	*Lev4*
4	1	1	Math	1	1	1	1
4	1	2	Physics	1	1	2	2
4	1	3	CS	1	1	3	3

To find a match for the first tuple above, and since L_b=3, we use the index structure to probe the H_3 branch index, which is based on columns (*Lev3*,*Level*,*PerLv*), to retrieve all nodes that match (1,4,2) search criterion. Similarly, the second and the third tuples are matched by probing the same H_3 branch index for nodes that match (2,4,2) and (3,4,2) search criteria, respectively, and hence the following tuples below are returned. If there are multiple nodes that match a search criterion, we retrieve them by invoking only one index lookup because they are physically clustered together.

Program	***Fname***
Math	Omar
Physics	Ayah
CS	Sue

5.2.3 Size Optimization

The UISX system only maps leaf nodes because internal nodes can be regenerated using the *PathSummary* and the *LeafNodes* tables.

Claim 5.1. Suppose S is a path summary for an XML data-tree G, V_L is the set of leaf nodes of G, and T is the set of serial paths of V_L. Then we can use S and T to reconstruct the subtree that is rooted at any internal node v, where $v \in V_R$.

Next, we present Algorithm 5.1 that reconstructs a subtree that is rooted at an internal node v where $v \in V_R$. The algorithm is a proof for Claim 5.1 above, which establishes that a subtree rooted at any internal node can be reconstructed by using *S* and *T*.

Algorithm 5.1 : Publish an internal node

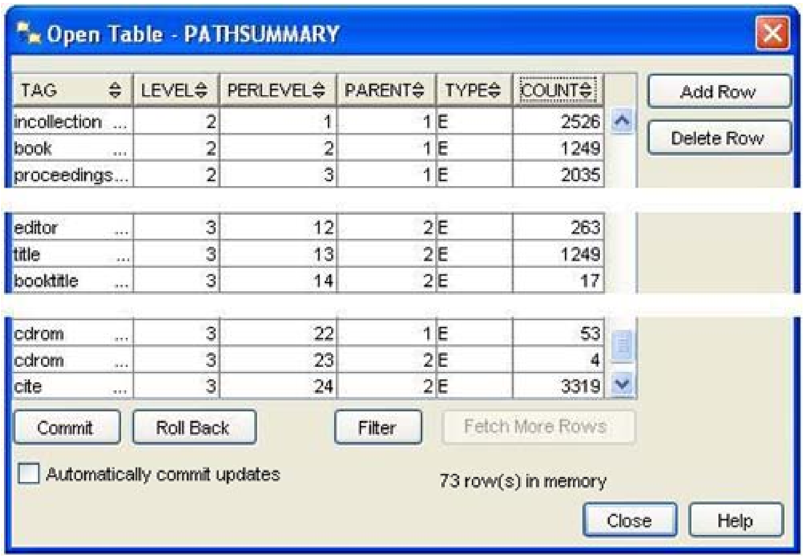
```
Input   :  An internal node $v$ .
Output  :  Subtree(s) rooted at $v$ .
1 Identify by using $S$:
    a- The branching node $v$ according to a given path,
    b- structure of the subtree $ST$ that is rooted at $v$ , and
    c- leaf nodes $L$ of $ST$, and sort them by $Level$ and $PerLev$ .
    $CheckedNodes$ = empty set { }
2 For each node $l$ in $L$
   Begin
    For $i = 1$ to $lc$ // $lc$ is the count of node $l$
     Begin
      While($CurrentNode.Level \le v.Level$ and $CurrentNode$ Not in $CheckedNodes$)
             Add $CurrentNode$ ($ChildNode, ParentNode$ ) to $CheckedNodes$
             $CurrentNode = CurrentNode.Parent$
     End
   End
3 Sort nodes in $CheckedNodes$ based on $<d,p,s>$.
  For each root node $v$ in $CheckedNodes$
   Begin
      $Subtree$ = empty tree { }
      $IdentifyChildren(v)$
        Begin
           Add $v$ to $Subtree$
           $ChildrenSet = \{v.child\}$
           For each child $y$ in $ChildrenSet$
             Begin
                if $y$ Is Not in $LeafNode$
                     $IdentifyChildren(y)$
                else
                     Add $v$ to $Subtree$
             End
        End
   End
  Return the $subtree$ rooted at $v$ node .
```

Step 1, as outlined in the left-hand side of Algorithm 5.1, identifies the internal node *v* that need to be published, the structure of the subtree *ST* rooted at *v*, and the leaf nodes *L* of *ST*. This step also initializes an empty set of checked nodes. Step 2 identifies all instances of all nodes that exist in *ST* and adds them to the temporary storage repository *CheckedNodes*. For each node, it adds the labels of the child (the current node) and the parent nodes, which are connected through an edge. Step 3 contains a recursive function that takes all nodes in the *CheckedNodes* repository and builds the subtrees that exist in these nodes according to parent-child relations using the labels obtained at the previous step. Note that step 2 follows a bottom-up tree traversal direction, while step 3 follows a top-down tree traversal direction. This algorithm is designed to reconstruct a subtree rooted at single *S* node that satisfies a query path. Adjustment to adapt to multiple *S* nodes that satisfy a given query path can be implemented by adding an outer loop to the algorithm to cover all satisfying nodes. All nodes *N* of *ST* are scanned and retrieved only once in which they are added to a temporary repository that are used at a subsequent step to rebuilt the original subtrees, and hence the complexity of this algorithms is *O(N)* database accesses in the worst case. Since nodes *N* are clustered by their *<d.p>* values, the actual database accesses are less than that expected by the worst case analysis. Next, we trace a simple example that shows how an internal node is published to demonstrate our claim.

Example 5.2. To illustrate how the reconstruction of an internal node is carried out, we use parts of the DBLP XML database that we use in our experimental evaluation. Table 5.3 represents a portion of the actual *PathSummary* table of the DBLP database that we implemented using DB2® V9.5.

Figure 5.6 illustrates a portion of the DBLP summary tree. The numbers below the elements' tags represent the count *C* of the extent nodes in the source XML database for the designated elements in the summary, which are taken from the *COUNT* field in the *PathSummary* table (Table 5.3). For simplicity, we assume in this example that the *book* element has only three child elements (*title*, *cdrom*, and *cite*).

Table 5.3 Part of the *PathSummary* of the 50 MB DBLP XML Database



Open Table - PATHSUMMARY

TAG	LEVEL	PERLEVEL	PARENT	TYPE	COUNT
incollection ...	2	1	1	E	2526
book ...	2	2	1	E	1249
proceedings...	2	3	1	E	2035
editor ...	3	12	2	E	263
title ...	3	13	2	E	1249
booktitle ...	3	14	2	E	17
cdrom ...	3	22	1	E	53
cdrom ...	3	23	2	E	4
cite ...	3	24	2	E	3319

73 row(s) in memory

To evaluate the query *"//book"* we have to reconstruct the internal node *book* as per the structure shown in Figure 5.6. We use the *PathSummary* and the *LeafNodes* tables to implement the reconstruction as follows.

- From the *PathSummary* table we can see that the *C* value of *book* element is 1249, in other words, there are 1249 instances of the *book* element, and these instances are associated by child relations with: 1249 instances of the *title* element, 4 instances of the *cdrom* element, and 3319 instances of the *cite* element. For repetitive referencing, we refer to the *book* element here as the parent element, and the *title*, *cdrom*, and *cite* elements as the child elements.

- At this stage we want to determine which child instances are associated with each parent instance. In order to show how to do that, we use the *LeafNodes* table. We take only the instances of the *cdrom* element in the *LeafNodes* table, which are shown in Table 5.4, as an example.
- Note that the parent element (the root element of the subtree) is located at level 2 (*Lb*=2) and the child elements are located at level 3 as shown in Figure 5.6. Also, from Table 5.4, we can see that the first instance (the first tuple in Table 5.4), whose *SerNo*=*1*, of the *cdrom* element is associated with instance number 4 (*s* value at *Lev2*) of the *book* parent element. Similarly, the second instance of *cdrom* element is associated with instance number 22 of the book parent element, and so on.

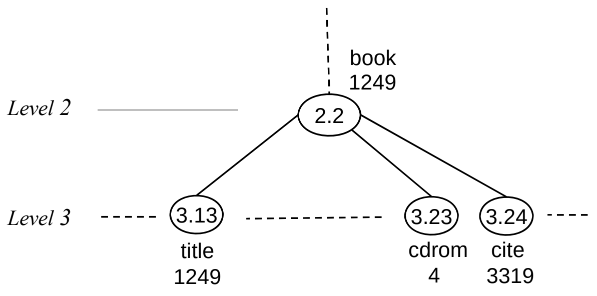


Figure 5.6 A portion of the 50 MB DBLP XML path summary tree

In this way we can reconstruct and publish the internal nodes. In our example in Figure 5.6, each instance of the *book* element has only one *title* child element. Just 4 instances of the *book* element have 4 instances of *cdrom* child element, in one-to-one relation. Finally, some instances of the *book* parent element have multiple instances of the *cite* child element.

Table 5.4 The tuples of *cdrom* element in the *LeafNodes* table

Level	*PerLevel*	*SerNo*	*Value*	*Lev1*	*Lev2*	*Lev3*	*Lev4*
3	23	1	AHV/Toc.pdf	1	4	1	0
3	23	2	BERNSTEIN/Contents.pdf	1	22	2	0
3	23	3	MAIER/CONTENTS.pdf	1	151	3	0
3	23	4	Wiederhold/toc.html	1	443	4	0

Since we can reconstruct internal nodes from the *LeafNodes* and the *PathSummary* tables, we do not need to store them, so the size of the mapped database can be reduced significantly. For example, the actual size savings in our experiments are 115 MB and 88 MB for the XMark and the DBLP databases, respectively (see Table 5.5). Another source of space saving in our approach is the fact that the paths of the nodes (elements and attributes) are not recorded in the database, as the case in other approaches [27] [37], because we can regenerate them from the *PathSummary* table by using the nodes' labels.

Table 5.5 Sizes of XMark and DBLP data-sets with different implementations

	Original size	*UISX with Internal nodes mapping*	*UISX without internal nodes mapping*	*Saved space*	*Percentage of saved space*
XMARK	100 MB	250	158	115	42%
DBLP	50 MB	155	85	88	51%

5.2.4 UISX Query Processor

This section discusses the components of the UISX query processor and the algorithm used in evaluating twig queries. We evaluate twig queries using a light-weight native XML engine on top of an SQL engine as illustrated in Figure 5.7. Hence we refer to this method as a *hybrid* query processor. The job of the native XML engine is to explore potential query optimization processes that are related to the structure of XML data, which can not be exploited by SQL engines. The SQL engine handles the XML-Relational data after shredding.

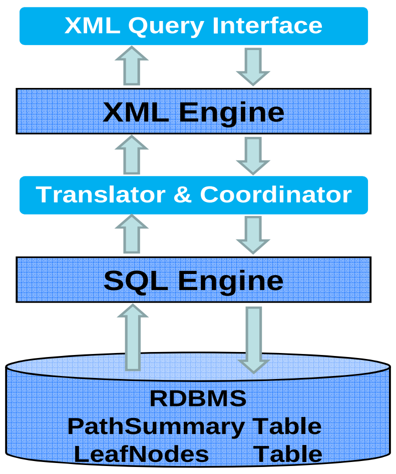


Figure 5.7 The UISX hybrid query processor

We developed Algorithm 5.2 to evaluate twig queries with one branching node. To evaluate a query with multiple branching nodes, the query is divided into several subtrees that are rooted at the branching nodes. The most nested subtree is evaluated first, and then the result is used to evaluate the subtree that is rooted at the next higher branching node, and so on.

The algorithm consists of 5 parts, which are indicated on the left-hand side of the algorithm. Please note that curly brackets stand for a set that can contain one or more node(s) or element(s); (*d*,*p*) represents the node in the *path summary* table whose *Level* is specified by *d* and *PerLv* is specified by *p*; and (*d*,*p*,*V*,*Lev*(*x*)) stands for the tuple in the *LeafNodes* table whose *Level* is *d*, *PerLv* is *p*, the *Value* of the tuple is *V*, and the *Lev*(*x*) represent the value of *Lev*(*x*) in the *LeafNodes* table where *x* is equal to the level of the branching node, namely *Lb*.

Step 1 of Algorithm 5.2 identifies the labels *d* and *p* (*Level*, and *PerLv*) of the leaf nodes for all branches in a twig query, in addition to the level of the branching node *Lb*. The second step identifies the branch with the minimal cardinality if no predicates are given in the query, where the cardinality can be identified from the *Count* field in the *PathSummary* table, or it identifies the branch with the higher selectivity if predicates are used in the query [13]. We need this step to minimize the number of nodes examined to identify a match, and hence reduce the evaluation cost. Step 3 in the algorithm identifies the set of secondary branches, which contains all the branches identified in step 1 minus the base branch identified in step 2. Step 4 evaluates the base branch by identifying the set of tuples that satisfy the predicates obtained in the preceding steps. These predicates include the values of *d*, *p*, and *x*. Note that the first two predicates are obtained from step 2, and the third predicate is obtained from step 1 where *x* (to be used in *Lev*(*x*)) is equal to *Lb*. We use these predicates to identify

V and $Lev(x)$ values of the leaf nodes of the base branch. Finally, in step 5, the information obtained in the previous steps - specifically d, p, and $Lev(x)$ values - is used to evaluate the secondary branches and obtain the final answers, which are returned to the user as the answer to the given query.

Algorithm 5.2: Evaluating twig queries

```
Algorithm to evaluate twig queries by using the hybrid method.
Input  :  Multiple paths XML query $Q$ .
Output :  Answer to all leaf nodes of the query .
1 //Use the PathSummary table to identify the Level & PerLev sets $\{(d_i,p_i)\}$
  //for all leaf nodes of all branches, plus the Level of the branching node $L_b$.
   $M(Q) \rightarrow \{(d_i,p_i)\}$ // mapping of $Q$ in $S$, identify leaf nodes
   $M(Q) \rightarrow (L_b)$      // mapping of $Q$ in $S$, identify the branching node level
2 //define the base branch $(d_{min},p_{min})$.
   $(d_{min},p_{min})=(d_1,p_1)$
   For $k=2$ to $i$
        if     Count$(d_k,p_k)$ < Count$(d_{min},p_{min})$
        then  $(d_{min},p_{min})=(d_k,p_k)$
3 //define the secondary branches $(d_r,p_r)$.
   $\{(d_r,p_r)\}=\{\{(d_i,p_i)\}-(d_{min},p_{min})\}$
4 //Evaluate the base branch first.
   use LeafNodes table to find set of tuples $\{(d_j,p_j,V_m,Lev(x))\}$
      where      $d_j=d_{min}$   and
                 $p_j=p_{min}$   and
                 $x=L_b$
5//Evaluate the secondary branches by using the base branch information .
   For each tuple in $\{(d_j,p_j,V_m,Lev(x))\}$ returned by step 4
    Begin
      For each branch in $\{(d_r,p_r)\}$
           Find $\{(d_r,p_r,V_n,Lev(y))\}$
              where   $y=L_b$ and
                      $Lev(x)=Lev(y)$
      Return $(V_m, \{\{V_n\}\})$
    End
```

In Algorithm 5.2 there are two factors that affect the number of accesses to indexes: the number of branches and the selectivity of the branches. Generally, the number of accesses to indexes is affected exponentially by the number of branches and linearly by the selectivity of the leaf nodes. The numbers of returned tuples and indexes accesses are inversely proportional to the selectivity of the leaf nodes of the branches in twig queries. False positive answers for a query are not possible with this algorithm since the set of retrieved tuples (candidate tuples) forms the exact answer to the query.

5.3 Prototype Implementation

To validate our approach, we implemented a prototype of the UISX in our lab using Java 1.6 and DB2® V9.5. We performed all experiments on a 3 GHz Intel® Pentium 4 PC running Windows® XP, with 1.5 GB of RAM. We use IBM's DB2® V9.5 RDBMS [64] to store and retrieve XML shredded data. The goal of the experiments is to evaluate the performance in terms of elapsed time to execute a query and the sizes of the databases, and the supporting indexes that are used by UISX system. We compare our approach with the approaches proposed by Chen et al. [27] for two reasons. First, they adopt a similar approach by creating branching nodes indexes that facilitate and guarantee one index lookup to find matches for each node returned by the base branch evaluation. Second, they compared their approach with five existing indexing schemes including: Edge table [88] and simulated DataGuide [54], which are based on edge model-mapping; simulated Index Fabric [37] and Access Support Relation (ASR) [71], which are based on forward-paths model-mapping; and Join Indices (JI) [125]. Chen et al. [27] proved experimentally that, in general, their approaches outperform these schemes.

5.3.1 Testing Data and Queries

Our experiments were carried out using the XMark [111], and DBLP [120] datasets. We used the test queries proposed by Chen et al. [27] because they are broad and cover different criteria such as cardinality, selectivity, recursion, and depth of the branch node. For ease of reference, the queries are listed in Table 5.6. Table 5.7 contains a summary of the characteristics of the test query sets in Table 5.6. The first set covers single path queries. The second set covers twig (multiple paths) queries with different selectivity and high branch points. The third set covers twig queries with low branch points. The fourth set covers recursive queries. The "X" and the "D" in the "Query No" column in Table 5.6 stand for the XMark and the DBLP databases, respectively.

Table 5.6 Four sets of queries used in testing

Set No	Qry No	Testing Query	Result per Branch
1	Q1X	/site/regions/namerica/item/quantity [. = 5]	1
	Q1D	/dblp/inproceedings/year [. = 1968]	1
	Q2X	/site/regions/namerica/item/quantity [. = 2]	709
	Q2D	/dblp/inproceedings/year [. = 1988]	1746
	Q3X	/site/regions/namerica/item/quantity [. = 1]	9228
	Q3D	/dblp/inproceedings/year [. = 2004]	10660
2	Q4X	/site [/people/person/profile/@income = 46814.17]	1
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q5X	/site [/people/person/profile/@income = 46814.17]	1
		[/people/person/name = ‘Hagen Artosi’]	1
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q6X	/site [/people/person/profile/@income = 9876.00]	2038
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q7X	/site [/people/person/profile/@income = 9876.00]	2038
		[/regions/namerica/item/location = ‘United States’]	7519
		/open_auctions/open_auction/bidder[/increase = 75.00]	55
	Q8X	/site [/people/person/profile/@income = 9876.00]	2038
		/open_auctions/open_auction/bidder[/ increase = 3.00]	5172
	Q9X	/site [/people/person/profile/@income = 9876.00]	2038
		[/regions/namerica/item/location = ‘United States’]	7519
		/open_auctions/open_auction /bidder[/increase = 3.00]	5172
3	Q10X	/site/open_auctions/open_auction	
		[/annotation/author/@person = ‘person22082’]	2
		/bidder/time	59486
	Q11X	/site/open_auctions/open_auction	
		[/annotation/author/@person = ‘person22082’]	2
		[/bidder/increase = 3.00]	5172
		/bidder/time	59486
4	Q12X	/site//item[/incategory/category = ‘category440’]	41
		/mailbox/mail/date	20946
	Q13X	/site//item[/incategory/category = ‘category440’]	41
		/mailbox/mail/date	20946
		/mailbox/mail/to	20946
	Q14X	/site//item[/quantity = 2]	1543
		[/location = ‘United States’]	16294
	Q15X	/site//item[/quantity = 2]	1543
		[/location = ‘United States’]	16294
		/mailbox/mail/to	20946

Table 5.7 Characteristics of the testing query sets in Table 5.6

Query Set	*Branches*	*Result Per Branch*	*Branch Depth*	*Recursion*
1	1	1 – 10660	N/A	0
2	2-3	1 – 7519	High	0
3	2-3	2 – 59486	Low	0
4	2-3	41 – 20946	Low	1

We executed each query ten times against its respective dataset and used the average of the 10 readings in our analysis. The time to translate the XPath queries to SQL queries is not included and only the execution times of the queries are recorded, which reflect the impact of the index structures.

5.3.2 Experimental Results

Table 5.8 summarizes the characteristics of the ROOTPATHS, DATAPATHS, and UISX index structures for the XMark and the DBLP databases. The tables and indexes sizes are in Megabytes. Note that the original sizes of XMark and DBLP datasets are 100 MB and 50 MB, respectively.

Table 5.8 Characteristics of testing databases as implemented by the indicated approaches

	ROOTPATHS	*DATAPATHS*	*UISX*
XMARK Tables Size MB	267	1,285	158
DBLP Tables Size MB	151	381	85
XMARK Indexes Size MB	509	2,535	325
DBLP Indexes Size MB	282	402	183
XMARK No. of Tuples	2,995,272	15,734,707	1,158,492
DBLP No. of Tuples	2,709,327	8,022,673	1,296,328

The ROOTPATHS and DATAPATHS indexes in Table 5.8 were not subjected to the compression methods listed in Chen et al.'s paper. To save the shredded data of the XMark database, UISX used a space equal to 59% of the space used by ROOTPATHS, and 12% of the space used by DATAPATHS. Similarly, to save the shredded data of the DBLP database, UISX used a space equal to 56% of the space used by ROOTPATHS, and 22% of the space used by DATAPATHS. With regard to the indexes size for XMark database, UISX used 64% of the space used by ROOTPATHS, and 13% of the space used by DATAPATHS. With regard to the indexes size of the DBLP database, UISX used 65% of the space used by ROOTPATHS, and 46% of the space used by DATAPATHS. Finally, the number of tuples that is required by UISX to shred the XMark XML database is equal to 39% of the tuples required by ROOTPATHS, and 7% of the tuples required by DATAPATHS; and for DBLP, UISX requires 48% of the number of tuples required by ROOTPATHS and 16% of the tuples required by DATAPATHS.

The results of the performance tests of UISX compared to ROOTPATHS and DATAPATHS with regards to the sets of test queries in Table 5.6 are illustrated in Figures 5.8-5.10. Note that *log*$_{10}$ scale is used to measure execution time.

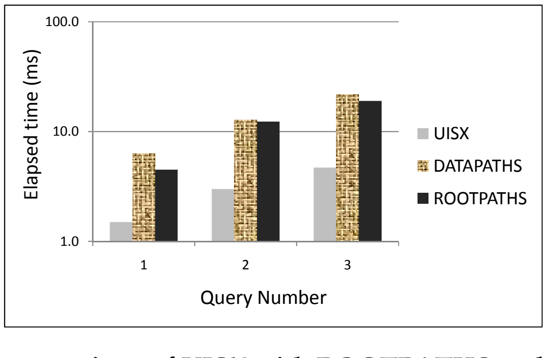


Figure 5.8 Performance comparison of UISX with ROOTPATHS and DATAPATHS using the DBLP database

We tested the DBLP data-tree with just the single branch queries since its depth is shallow. The results of the 3 test queries, given in Figure 5.8, show that UISX performs 67% - 76% better than the ROOTPATHS, and 76% - 79% better than DATAPATHS.

Figure 5.9 presents the elapsed execution time of the test queries with UISX compared with ROOTPATHS using the XMark database. We notice that the gain in performance is fairly steady (53% – 64%) for the first type (single branch queries 1-3). On the other hand, the percentage gain in performance for the second type of queries (queries 4 – 9) decreases from the 81% to 31% as the selectivity decreases, since the number of pages that contain the returned tuples of the elements with high selectivity are smaller than those with low selectivity. The gain in performance for the third type of test queries is extremely high (99%) because with ROOTPATHS, each tuple returned by the base branch evaluation result has to be hash-joined or merged with the tuples returned by the secondary branches in order to find the matching tuples. While in UISX, the matching tuples of each secondary branch are retrieved with one comparison for each tuple returned by the base branch. The gain in performance for the fourth type of queries is high (79% - 89%) for the selective queries (queries 12 and 13), and relatively low (19% - 37%) for the queries with low selectivity (queries 14 and 15).

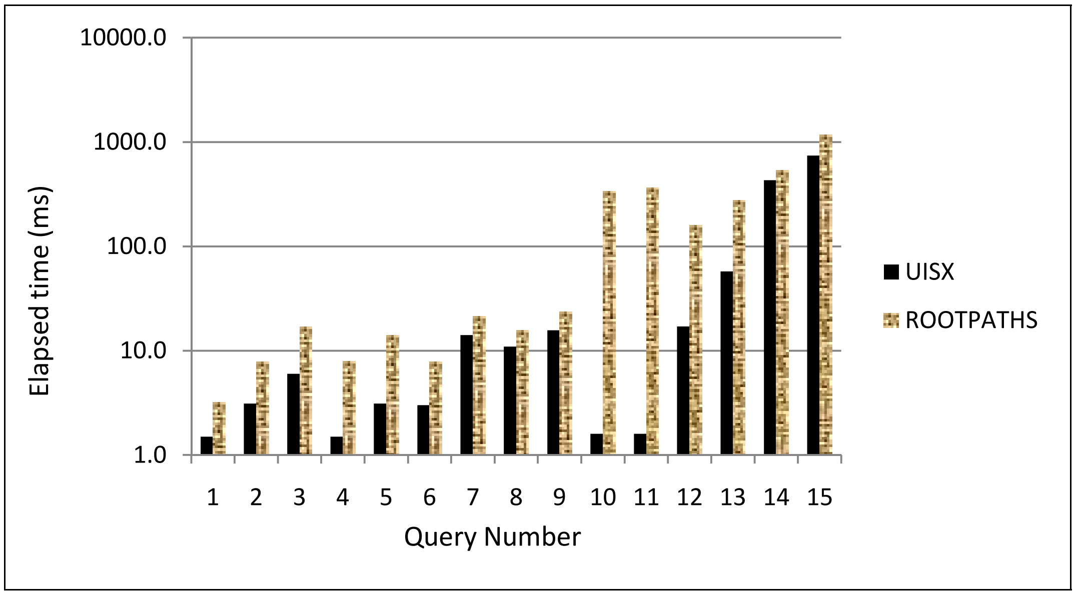


Figure 5.9 The performance comparison of UISX with ROOTPATHS using XMark database

Figure 5.10 shows the comparison of UISX with DATAPATHS using the XMark database. We notice that the gain percentage for the first type of queries is steady and it is in the fifties. Similar to the performance tests against the ROOTPATHS, the gain percentage in the performance of the second type of queries decreases as the selectivity decreases and ranges between (9% - 53%). The gain percentage in performance for the third type is ranging between (48% - 50%). Also, the gain percentage in performance for the first two queries of the forth type of queries (queries with high selectivity) ranges between (36% - 44%), which is higher than that of the last two queries (queries with low selectivity), which ranges between (9% -24%).

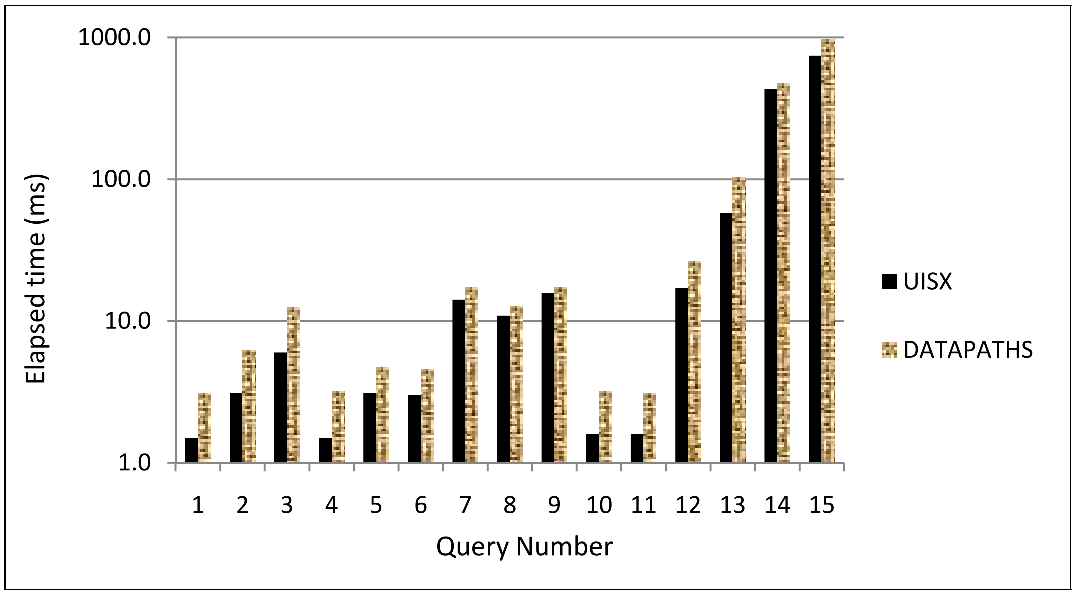


Figure 5.10 The performance comparison of UISX with DATAPATHS using XMark database

Our approach performs well in comparison to ROOTPATHS and DATAPATHS. Generally, we believe that the UISX performance gains over ROOTPATHS are mainly because UISX does not produce any false positive answers, while ROOTPATHS does. DATAPATHS does not produce any false positive, and it is an efficient index structure, but its size is large. UISX performance gains over DATAPATHS are due to the relatively small size of the UISX index structure. Larger indexes require a deeper B+-tree, and hence require more search. An efficient way to evaluate a query using DATAPATHS index structure is by evaluating the base branch first. Then a mechanism has to be implemented in order to extract the ids of the branching nodes from the returned *IdLists* (e.g. scan the *IdList* string by implementing a string matching operations).

In UISX, in contrast, when the base branch is evaluated, the branching nodes ids (labels) are returned in the fields (*Lev2,Lev3,Lev4,..etc*), and are ready to be used in matching operations directly without the need for extra techniques to extract the branching nodes ids.

To evaluate recursive queries (queries 12-15), the reversed-path approaches use the Optional String Pattern Matching (OSPM) function ("LIKE") to evaluate a path with ancestor-descendent axis [56]. For example, if we assume that *S* and *A* stands for *student* and *address* elements, respectively, then the query *"//student/address"* would be evaluated by using an SQL query that would contain the statement *"SchemaPath LIKE AS%"* along with other statements. In contrast, UISX approach uses only the exact string pattern matching ("="). For example to find the nodes that match the path in the above query, we would run the following SQL query:

```
Select  s1.level, s1.perlv
From    PathSummary as s1, PathSummary as s2
Where   s1.tag='address'           and
        s2.tag='student'           and
        s1.parent=s2.perlv
```

It is known that SQL supports exact string pattern matching efficiently by using the B+-tree indexes, while B+-tree indexes does not support ("LIKE") efficiently [56].

5.4 Summary

Twig queries can be evaluated by using knowledge of their branching nodes. We propose an approach that utilizes this idea to evaluate twig queries efficiently by building a universal index structure that covers all nodes in XML databases. This index structure guarantees to find a complete and accurate match in a secondary branch for each node of any arbitrary base branch by executing a single index lookup.

Our proposed approach has a compact size, yet, it supports efficient evaluations of twig queries. It uses a RDBMS to store and query XML documents. It also uses path summaries, which are based on DataGuides [54], to facilitate a query evaluation. The path summary, which is modeled as a simple table in a relational database, reduces the number of matches required to evaluate a query by preserving a path summary of the original XML data structure before shredding. Path summaries reduce the size of the stored XML databases. This reduction in size is achieved by: (1) eliminating redundant data from the database, such as the path of an element, which can be regenerated when needed from the summary, and (2) by using the summary to regenerate the internal nodes of the XML data-tree along with their subtrees. Therefore, internal nodes do not need to be shredded and stored in relational tables. In our approach, only the leaf nodes are shredded and stored in relational tables. The root-paths are recorded for all leaf nodes, where the information of the internal nodes is encoded.

Zhang et al. [143] observed that RDBMSs do not support the inequality-joins efficiently, while they support the equality-joins efficiently. Our XML-relational approach evaluates XML queries by using equijoins, while most XML-relational approaches use inequality-joins [56] [143].

We use light-weight native XML engine on top of an SQL engine to evaluate queries. The job of the native XML engine is to explore potential query optimization processes that are related to the structure of XML data, which can not be exploited by SQL engines. The SQL engine handles the XML-Relational data after shredding.

We implement the UISX index structure in our laboratory successfully using the DB2® V9.5 DBMS [64], and the experimental results show that it performs well in comparison to existing state-of-the-art approaches in terms of size and response time.

Chapter 6

Conclusions

XML employs a tree-structured data model. Therefore, an XML query typically consists of two parts: structure constraints and values. Since the repetition of XML data is irregular due to missing and/or repeated arbitrary elements, its storage structure can be scattered over many different locations on the disk, which decreases the performance of XML queries [32]. Furthermore, the flexibility of specifications of the XML queries (e.g. use of wild cards) adds to the challenge of indexing methods [130] [146]. These complexities offer many new challenges for the researchers and software vendors. In this dissertation we present a labelling scheme and two index structures for XML databases. We conclude our dissertation in this chapter by providing a summary of the previous chapters and discussing some of the future work directions.

6.1 Summary

In Chapter 2 we give a brief history of the creation and the development of the XML data model. Then we discuss the three main categories of structural indexing schemes proposed in the literature to handle the XML semistructured data model. Finally, we discuss limitations and open problems related to the major existing indexing schemes. We classify XML structural indexes according to two important characteristics: determinism and bisimilarity, since these characteristics controls the size of indexes and their query answering power.

Two of the most widely used labeling schemes are interval and prefix labeling schemes. Each type of scheme has advantages and disadvantages. We design a labeling scheme that has the advantages of the two types of schemes while eliminating the main disadvantages in Chapter 3. This labeling scheme is based on the levels of elements in XML trees.

There are two methods for storing and querying XML databases. The first method relies on native hierarchical nesting structure of XML databases. The other method leverages the existing power of RDBMSs that has been established over several decades. We design a native and an XML-Relational index structures in Chapters 4 and 5, respectively.

We address the excessive number of joins and match operations required to evaluate a query in Chapter 4 and we propose an index structure that remarkably reduces them. The proposed index structure also eliminates a great number of search space that is associated with XML data model.

Generally there is a tradeoff between the size and the power of indexes [56] [144]. The state-of-the-art XML structural indexes suffer from either being huge to perform well or perform poorly as a consequence to saving size. Chapter 5 presents an indexing approach that minimizes the size of XML structural indexes, yet performs well. This approach is based on two ideas: indexing the branching nodes of XML trees and map XML data into relational tables using a novel mapping scheme that is based on graph summary mapping. In the following section we present our future work directions before concluding the dissertation.

6.2 Future Directions and Challenges

The main challenge in indexing XML data is the irregularity of data and structure. Value-based queries can be evaluated by using traditional indexing schemes, such as B+-trees or inverted lists. However, efficient support for the structural part is a challenging task. The semistructured nature of XML data and the flexibility of the query languages pose another distinctive concern for deriving or selecting proper indexing methods. Designing representations for efficient storage of semistructured data is also a difficult task.

Making the existing labeling schemes – including the LLS – dynamic so that they adapt gracefully to deletion and insertion of new nodes is not an easy task. Choosing an appropriate index definition that covers a given query workload is an open problem for $(F+B)^k$-index. Also, efficient index building and updating algorithms are needed for non-deterministic forward and backward bisimilar indexes. Efficient integration of graph indexes with value indexes is another interesting area. This will minimize the I/O accesses by eliminating the need to access two different indexes to evaluate an XML query with a predicate. A hierarchy of graph covering indexes is yet another open area of research. The hierarchies could be defined in terms of summary tables, where higher level summaries could be extracted from lower level summary tables.

Sequence indexes support solving a twig query only in a certain order. If the query order does not match the index order it will return an incorrect answer. To run a query against a sequence index all possible orders of the query nodes have to be tested in order to get an accurate result. The node and graph indexes do not have this problem. Another limitation of sequence indexes is that they may require a large number of accesses to the index, consequently, it might result in expensive random I/O accesses. Finally, the overhead of the false positives problem is a major drawback of sequence indexes.

One of the challenges we are planning to pursue is to identify a suitable set of statistics for a given graph-based data that can be efficiently computed and stored without having a fixed graph index. We are also planning to use a customized XML storage media for the data in LTIX system, instead of using B+-tree storage media. We would like to extend our relational indexing scheme by adding a module to the XML query engine that translates XPath queries into SQL queries where the hierarchy of XML paths is reflected properly.

Finally, we are planning to improve on the native XML query engine that works on top of the SQL engine in UISX system. We think that coordinating the query optimization tasks between these two engines can improve XML query processing. Despite the fact that the size of the shredded data in UISX system is minimized, since the leaf nodes contain details about both themselves and the internal nodes, we noticed that some of these details are redundant among multiple leaf nodes. For example, leaf nodes that share the same branching node have similar information about the path from the root node to the branching node. It worth investigating if there is a way to eliminate these redundancies and improves performance at the same time.

Trademarks

References

[1] Abiteboul. S., 1997. Querying semistructured data. In *Proc. of the International Conference on Database Theory, ICDT'97,* Delphi, Greece, LNCS 1186, pp. 1–18.

[2] Abiteboul, S., Buneman, P., and Suciu, D., 2002. *Data on the Web: From Relations to Semistructured Data and XML,* San Francisco, California, USA: Morgan Kaufmann Publishers.

[3] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and .Wiener. J., 1997. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1), 68-88.

[4] Aboulnaga, A., Alameldeen, A., and Naughton, J., 2001. Estimating the Selectivity of XML Path Expressions for Internet Scale Application. In *Proc. of the 27th VLDB Conference*, Roma, Italy, pp. 591-600.

[5] Ali, M.S., Consens, M., Gu, X., Kanza, Y., rizzolo, F., and Stasiu, R., 2007. Efficient, Effective and Flexible XML Retrieval Using Summaries. In *Comparative Evaluation of XML Information Retrieval Systems*, LNCS 4518, pp. 89-103.

[6] Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J., Srivastava, D., and Wu, Y., 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the 18th ICDE*, pp. 141-154.

[7] Alstrup, S., Bille, P., and Rauhe, T., 2003. Labeling Schemes for Small Distances in Trees. In *Proc. of the 14th Annual ACM-SIAM Symposium on Discrete Mathematics*, Baltimore, Maryland, USA, pp. 689-698.

[8] Amagasa, T., Yoshikawa, M., and Uemrua, S., 2003. QRS: A Robust Numbering Scheme for XML Documents. In *Proc. of the 19th International Conference on Data Engineering,* Bangalore, India, pp. 705-707.

[9] Amer-Yahia, S., Baeza-Yates, R., Consens, M., and Lalmas, M., 2007. XML Retrieval: DB/IR in Theory, Web in practice. In *Proc. of the 33rd International Conference on Very Large Databases,* Vienna, Austria, pp. 1437-1438.

[10] Angles, R. and Gutierrez, C., 2008. Survey of Graph Database Models. *ACM Computing Surveys*, 40(1), Article No.1.

[11] Baeza-Yates, R., Consens, M., 2004. The Continued Saga of DB-IR Integration. In *Proc. of the 30th VLDB Conference*. Toronto, Canada, pp. 1245-1246.

[12] Barta, A., Consens, M., and Mendelzon, A., 2004. XML Query Optimization Using Graph indexes. In *Proc. of the 1st International Workshop on XQuery Implementation, Experience, and Perspectives, in cooperation with ACM SIGMOD,* Paris, France, pp. 43-48.

[13] Barta, A., Consens, M., and Mendelzon, A., 2005, Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In *Proc. of the 31st VLDB Conference*, Trondheim, Norway, pp. 133-144.

[14] Bary, T., Paoli, J., and Sperberg-McQueen, C. (Eds.). 1998. *Extensible Markup language (XML) 1.0*. Retrieved January 22, 2009, from http://www.w3.org/TR/1998/REC-xml-19980210.html.

[15] Bertino, E., Rabitti, F., and Gibbs. S., 1998. Query Processing in a Multimedia Document System. A*CM Transactions on Office Information Systems,* 6(1), 1–41.

[16] Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Simeon, J. (Eds.). 2007. *XQuery 1.0: An XML Query Language*. Retrieved January 19, 2009, from http://www.w3.org/TR/xquery.

[17] Bonifati, A., Ceri, S., 2000. Comparative Analysis of Five XML Query Languages. *ACM SIGMOD Record*, 29(1), pp. 68-79.

[18] Bruno, N., Koudas, N., and Srivastava, D., 2002. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the ACM SIGMOD.* pp. 310-321.

[19] Buneman, P., 1997. Semistructured data. *In Proc. of the 16th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems,* Tucson, Arizona, USA, pp. 117–121.

[20] Buneman, P., Davidson, S., Hillebrand, G., and Suciu. D., 1996. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD International Conference on Management of Data,* Montreal, Quebec, Canada, pp. 505-516.

[21] Carey, P., 2004. *New Perspective on XML-Comprehensive.* Boston, Massachusetts, USA: Course Technology.

[22] Catania, B., Maddalena, A., and Vakali, A., 2005. XML Document Indexes: A Classification. *IEEE Internet Computing*, vol. (9)5, pp. 64-71.

[23] Catania, B., Ooi, B., Wang, W., and Wang, X., 2005. Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA, pp. 515-526.

[24] Chamberlain, D., Robie, J., and Florescu, D., 2000. Quilt: An XML Query Language for Heterogeneous Data Sources. In *The World Wide Web and Databases, 3rd International Workshop, WebDB'00,* Dallas, Texas, USA, LNCS 1997, pp. 1-25.

[25] Chaudhuri, S., and Shim, K., 2003. Storage and Retrieval of XML Data Using Relational Database. In *Proc. ICDE'03*, p. 802.

[26] Che, D., Aberer, K., and Ozsu, M., 2006. Query optimization in XML Structured-document Databases. *The VLDB Journal*, 15(3), 263-289.

[27] Chen, Z., Gehrke, J., Korn, F., Koudas, N., Shanmugasundaram, J., Srivastava, D., 2007. Index Structures for Matching XML Twigs Using Relational Query Processors. *Data & Knowledge Engineering*, 60(2), 283-302.

[28] Chen, Q., Lim, A., and Ong, K., 2003. D(k)-Index: An adaptive Structural summary for graph-structured data. *In Proc. of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, pp. 134-144.

[29] Chen, Y., Mihaila, G.A., Bordawekar, R., and Padmanabhan, S., 2004. L-Tree: a Dynamic Labeling Structure for Ordered XML Data. In *Current Trends in Database Technology – EDBT'04 Workshops,* Herakleion, Greece, LNCS 3268, pp. 209-218.

[30] Chien, S., Vagena, Z., Zhang, D., Tsotras, V., and Zaniolo, C., 2002. Efficient Structural Joins on Indexed XML Documents. In *Proc. of 28th ICDE,* pp. 263-274.

[31] Christophides, V., Plexousakis, D., Scholl, M., and Tourtounis, S., 2003. On Labeling Schemes for the Semantic Web. In *Proc. of the 12th International conference on World Wide Web*, Budapest, Hungary, pp. 544-555.

[32] Chung, C., Min, J., and Shim, K., 2002. APEX: An Adaptive Graph index for XML data. *In Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, pp. 121-132.

[33] Clark, J., and DeRose, S. (Eds.). 1999. *XML Path Language (XPath) Version 1.0.* Retrieved January 22, 2009, from http://www.w3.org/TR/xpath.

[34] Cohen, E., Kaplan, H., and Milo, T., 2002. Labeling Dynamic XML Trees. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems,* Madison, Wisconsin, USA, pp. 271-281.

[35] Combi, C., Lavarini, N., and Oliboni, B., 2006. Querying Semistructured Temporal Data. In *EDBT 2006 Workshops*, LNCS 4254, p. 625-636.

[36] Consens, M., Rizzolo, F., and Vaisman, A., 2008. AxPRE Summaries: Exploring the (Semi-) Structure of XML Web Collections. In *Proc. of the 24th International Conference on Data Engineering, ICDE'08*, Cancun, Mexico, pp. 1519-1521.

[37] Cooper, B., Sample, N., Franklin, M., Hjaltason, G., and Shadmon. M., 2001. A Fast Index for Semistructured Data. In *Proc. of 27th International Conference on Very Large Databases (VLDB),* Roma, Italy, pp. 341-350.

[38] De Aguiar, J., Filho, M., and Harder, T., 2006. Statistics for Cost-Based XML Query Optimization. In *18th GI-Workshop on the Foundations of Databases (Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken)*, Wittenberg, Sachsen-Anhalt, Germany: Institute of Computer Science, Martin-Luther-University, pp. 110-114.

[39] Deutsch, A., Fernandez, M., Florescu, D., Levy, A., and Suciu, D., 1998. *XML-QL: A query language for XML*. Retrieved January 20, 2009, from http://www.w3.org/TR/NOTE-xml-ql.

[40] Dietz. P. 1982. Maintaining order in a linked list. In *Proc. of the 14th annual ACM symposium on Theory of Computing,* San Francisco, California, USA, pp. 122 –127.

[41] Dong, X., and Halevy, A., 2007. Indexing Dataspaces. In *Proc. of the ACM SIGMOD International Conference on Management of Data,* Beijing, China, pp. 43-54.

[42] Duong, M., and Zhang, Y., 2005. LSDX: A New Labeling Scheme for Dynamically Updating XML Data. In *Database Technologies 2005, Proc. of 16th Australasian Database Conference*, Newcastle, Australia, Vol.39, pp. 185-193.

[43] Dweib, I., Awadi, A., Elrhman, S., and Lu, J., 2008. Schemaless Approach of Mapping XML Document into Relational Database. In *Proc. of CIT'*08, pp. 167-172.

[44] Elghandour, I., Abolnaga, A., Zilio, D., Chiang, F., Balmin, A., Beyer, K., and Zuzarte, C., 2008. An XML Index Advisor for DB2. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Vancouver, Canada, pp. 1267-1270.

[45] Feinberg, G., 2004. Anatomy of a Native XML database. In *XML 2004 Conference & Exhibition*, Washington, D.C., USA. Retrieved January 13, 2009 from http://www.idealliance.org/proceedings/xml04/ abstracts/paper170.html.

[46] Fernandez, M., Florescu, D., Kang, J., Levy, A., and Suciu. D., 1998. Catching the Boat with Strudel: Experiences with a Website Management System. In *Proc. ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA, pp. 414– 425.

[47] Fiebig, T., Helmer, S., Kanne, C., Moerkotte, G., Neumann, J., Schiele, R., et al., 2002. Anatomy of a Native XML Base Management System. *The VLDB Journal*, 11(4), 292-314.

[48] Fisher, D., Lam, F., Shui, W., and Wong, R., 2006. Dynamic Labeling Schemes for Ordered XML Based on Type Information. In *Proc. of the 17th Australasian Database Conference,* Hobart, Australia, Vol. 49, pp. 59-68.

[49] Florescu, D., and Kossmann, D., 1999. Storing and Querying XML Data Using an RDMBS. *Bulletin of the Technical Committee on Data Engineering,* 22(3), 27-34.

[50] Freire, J., and Benedikt, M., 2004. Managing XML Data: An Abridged Overview. *Computing in Science & Engineering, IEEE*, 6(4), 12-19.

[51] Fujimoto, K., Kha, D., Yoshikawa, M., and Amagasa, T., 2005. A Mapping Scheme for XML Documents into Relational Databases using Schema-based Path Identifiers. In *Proc. of the International Workshop in Web Information Retrieval and Integration*, Tokyo, Japan, pp. 82-90.

[52] Gardarin, G., Gruser, J., and Tang, Z., 1996. Cost-based Selection of Path Expression Processing Algorithms in Object-oriented Databases. In *Proc. of the 22nd International Conference on Very Large Databases,* Bombay, India, pp. 390–401.

[53] Goldman, R., McHugh, J., and Widom, J., 1999. From semistructured data to XML: Migrating the Lore data model and query language. In *Proc. of the 2nd International Workshop on the Web and Databases, ACM SIGMOD Workshop,* Philadelphia, Pennsylvania, USA, pp. 25-30.

[54] Goldman, R., and Widom, J., 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of 23rd International Conference on Databases, VLDB'97,* Athens, Greece, pp. 436-445.

[55] Goldman, R., and Widom, J., 1999. Approximate DataGuide. In *Proc. of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel.

[56] Gou, G., and Chirkova, R., 2007. Efficiently Querying Large XML Data Repositories: A Survey. *Transactions on Knowledge and Data Engineering,* 19(10), 1381-1403.

[57] Grust, T., 2002. Accelerating XPath Location Steps. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison. Wisconsin, USA, pp. 109-120.

[58] Guo, L., Shao, F., Botev, C., and Shanmugasundaram, J., 2003. XRANK: Ranked Keyword Search over XML Documents. *In Proc. of the ACM SIGMOD International Conference on Management of Data*. San Diego, California, USA, pp. 16-27.

[59] Halverson, A., Burger, J., Galanis, L., Kini, A., et al., 2003. Mixed Mode XML Query Processing. In the *Proc. of the 29th International Conference on VLDB*, Berlin, Germany, pp. 225-236.

[60] Harder, T., Haustein, M., Mathis, C., and Wagner, M., 2007. Node Labeling Schemes for Dynamic XML Documents Reconsidered. *Data & Knowledge Engineering*, 60(1), pp. 126-149.

[61] Harding, P., Li, Q., and Moon, B., 2003. XISS/R: XML Indexing and Storage System Using RDBMS. In *Proc. of VLDB*, pp. 1073-1076.

[62] Haw, S. and Lee, C., 2008. Evolution of Structural Path Indexing Techniques in XML Databases: A Survey and Open Discussion. In *Proc. of the 10th International Conference on Advanced Communication Technology*, Gangwon-Do, pp. 2054-2059.

[63] Haw, S., and Lee, C., 2009. Extending Graph Index and Region Encoding for Efficient Structural Query Processing in Native XML Databases. *The Journal of Systems and Software*. 82(2009): 1025-1035.

[64] IBM homepage on DB2. [Online]. Available: http://www-01.ibm.com/ software/data/db2/linux-unix-windows/.

[65] Jiang, H., Lu, H., Wang, W., and Chin Ooi, B., 2003. XR-tree: Indexing XML Data for Efficient Structural Joins. In *Proc. of the 19th International Conference on Data Engineering.* Bangalore, India, pp. 253-263.

[66] Kaplan, H., Milo, T., and Shabo, R., 2002. A Comparison of Labeling Schemes for Ancestor Queries. In *Proc. of the 13th annual ACM-SIAM Symposium on Discrete Algorithms*, San Fransisco, CA, USA, pp. 954-963.

[67] Kaushik, R., Bohannon, P., Naughton, J., and Korth, H., 2002. Covering Indexes for Branching Path Queries. *In Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, pp. 133-144.

[68] Kaushik, R., Bohannon, P., Naughton, J., and Shenoy, P., 2002. Updates for Structure Indexes. In *Proc. of 28th International Conference on Very Large Databases,* Hong Kong, China, pp. 239-250.

[69] Kaushik, R., Krishnamurthy, R., Naughton, J., and Ramakrishnan, R., 2004. On the Integration of Structure Indexes and Inverted Lists. In *Proc. of the ACM SIGMOD ICMD*, pp. 779-790.

[70] Kaushik, R., Shenoy, P., Bohannon, P., and Gudes, E., 2002. Exploiting Local Similarity for Indexing Paths in Graph-structured Data. In *Proc. of 18th International Conference on Data Engineering,* San Jose, California, USA, pp. 129-140.

[71] Kemper, A., and Moerkotte, G., 1990. Access Support in Object Bases. In *Proc. of SIGMOD*, pp. 364-374.

[72] Kha, D., Yoshikawa, M., Uemura, S., 2001. An XML Indexing Structure with Relative Region Coordinate. In *Proc. of the 17th International Conference on Data Engineering*, Heidelberg, Germany, pp. 313-320.

[73] Knuth. D., 1998. *The Art of Computer Programming: Vol. III. Sorting and Searching,* Reading, MA., USA: Addison-Wesley, 3rd ed., pp. 492-507.

[74] Kobayashi, M. and Takeda, K., 2000. Information Retrieval on the Web. A*CM Computer Surveys*, 32(2), pp. 144-173.

[75] Krishnamurthy, R., Kaushik, R., and Naughton, J., 2003. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Database and XML Technologies, First International XML Database Symposium, XSym'03*, Berlin, Germany, LNCS. 2824, pp. 1-18.

[76] Li, H., Lee, M., Hsu, W., and Chen, C., 2004. An Evaluation of XML Indexes for Structural Join. *ACM SIGMOD Record*, 33(3), pp. 28-33.

[77] Li, C. and Ling, T., 2005. QED: A Novel Quaternary Encoding to Completely Avoid Re-labeling in XML Updates. In *Proc. of the 14th ACM International Conference of Information and Knowledge Management*, Bremen, Germany, pp. 501-508.

[78] Li, Q., and Moon, B., 2001. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of 27th International Conference on Very Large Databases,* Roma, Italy, pp. 361-370.

[79] Li, Y., Yi, P., and Li, Q., 2005. Optimizing Path Expression Queries of XML Data. In *Proc. of the IEEE International Conference on e-business Engineering, ICEBE'05*, Beijing, China, pp. 497-504.

[80] Lu, J., and Ling, W., 2004. Labeling and Querying Dynamic XML Trees. In *Advanced Web Technologies and Applications, 6th Asia-Pacific Web Conference,* Hangzhou, China, LNCS 3007, pp. 180-189.

[81] Lu, J., Ling, T., Chan, C., and Chen, T., 2005. From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proc. of the 31st International Conference on Very Large Databases. VLDB*, Trondheim, Norway, pp. 193-204.

[82] Luk, R., Leong, H., Dillon, T., Chan, A., Bruce, W., and Allan, J., 2002. A Survey in Indexing and Searching XML Documents. *Journal of the American society for Information Science and Technology*, 53(6), pp. 415-437.

[83] Lv, J., Wang, G., Yu, J., and Yu, G. 2002. Performance Evaluation of a DOM-Based XML Database: Storage, Indexing, and Query Optimization. In *Advances in Web-Age Information Management, 3rd International Conference, WAIM'02*, Beijing, China, LNCS 2419, pp. 13-24.

[84] Mariano, P., and Baeza-Yates, R., 2005. Database and Information Retrieval Techniques for XML. In *Advances in Computer Science-ASIAN, Data Management on the Web, 10th Asian Computing Science Conference*, Kunming, China, LNCS 1318, pp. 22-27.

[85] Megginson, D., and Brownell, D., 2004. *Simple API for XML (SAX).* Retrieved January 22, 2009, from http://www.saxproject.org/.

[86] Mendelzon, A., Rizzolo, F., and Vaisman, A., 2004. Indexing Temporal XML Documents. In *Proc. of the 30th VLDB Conference*. Toronto, Canada, pp. 216-227.

[87] McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J., 1997. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record,* 26(3), 54-66.

[88] McHugh, J., and Widom, J,. 1999. Query Optimization for XML. In *Proc. of 25th International Conference on Very Large Databases, VLDB'99,* Edinburgh, Scotland, UK, pp. 315-326.

[89] Miler, J. and Sheth, S., 2000. Querying XML Documents. *IEEE Potentials Magazine,* 19(1), pp. 24-26.

[90] Milo, T., and Suciu, D., 1999. Index Structures for Path Expressions. In *Database Theory – ICDT'99, Proc. of 7th International Conference on Database Theory*, Jerusalem, Israel, LNCS 1540, pp.277-295.

[91] Min, J., Kim, J., and Lee, M., 2005, Effective Path Indexes for XML data on Relational Databases. In *Proc. of the 7th International Conference on Advanced Communication Technology*, Phoenix Park, Korea, vol. 2, pp. 1355-1359.

[92] Mohammad, S., and Martin, P., 2009. XML Structural Indexes (*Technical Report No. 2009-560*). Kinston, Ontario, Canada: Queen's University.

[93] Mohammad, S., and Martin, P., 2009. Index Structures for XML Databases. In Li, C., and Ling, T. W. (Eds.). *Advanced Applications and Structures in XML processing: Label Streams, Semantics Utilization and Data Query Technologies*. IGI Global. pp. 98-124

[94] Mohammad, S., and Martin, P., 2010. LLS: A Level-based Labeling Scheme for XML Databases. In *Proc. of CASCON 2010*, Toronto, Canada. pp. 115-127.

[95] Mohammad, S., and Martin, P., 2010. LTIX: A Compact Level-based Tree to Index XML Databases. In *Proc. of International Database Engineering and applications Symposium*, Montreal, Canada, pp. 21-25.

[96] Mohammad, S., Martin, P., and Powley, W., 2011. Relational Universal Index Structure for Evaluating XML Twig Queries. Accepted for publication in *Proc. of the International Conference on Communications and Information Technology – ICCIT 2011*, Aqaba, Jordan.

[97] Moro, M., Vagena, Z., and Tsotras, V., 2005. Tree-Pattern Queries on a Lightweight XML Processor. In *Proc. of the 31st VLDB Conference*, pp. 205-216.

[98] Naughton, J., DeWitt, D., Maier, D., Aboulnaga, A., Chen, J., Galanis, L., et al., 2001. The Niagara Internet Query System. *Bulletin of the Technical Committee on Data Engineering,* 24(2), 27-33.

[99] O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., and Westbury, N., 2004. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the ACM SIGMOD International Conference on Management of Data,* Paris, France, pp. 903-908.

[100] Online Computer Library Center. 2008. *Dewey Decimal Classification*. Retrieved January 13, 2009, from http://www.oclc.org/dewey/versions/ ddc22print/intro.pdf.

[101] Pal, S., Cseri, I., Seeliger, O., Schaller, G., Giakoumakis, L., and Zolotov, V., 2004. Indexing XML Data Stored in a Relational Database. In *proc. of VLDB*, pp. 1146-1157.

[102] Paparizos, S., Jagadis, H., Patel, J., Al-Khalifa, S., Ladshmanan, L., Srivastava, D., et al., 2003. TIMBER: A Native System for Querying XML. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 672-672.

[103] Polyzotis, N., Garofalakis, M., and Ioannidis, Y., 2004. Approximate XML Query Answers. In *Proc. of the ACM SIGMOD International Conference on Management of Data,* Paris, France, pp. 263-274.

[104] Rao, P., and Moon. B., 2004. PRIX: Indexing and Querying XML Using Prufer Sequences. In *Proc. of the 20th International Conference on Data Engineering, ICDE 2004,* Boston, MA, USA, pp. 288-300.

[105] Robie, J. (Ed.), Derksen, E., Fankhauser, P., Howland, E., Huck, G., Macherius, I., et al., 1999. *XML query language (XQL).* Retrieved January 20, 2009, from http://www.ibiblio.org/xql/xql-proposal.html.

[106] Sahuguet, A., 2000. *Kweelt, the Making-of: Mistakes Made and Lessons Learned* (Tech. Rep. No. MS-CIS-00-23). Pennsylvania, USA: University of Pennsylvania, Department of Computer and Information Science.

[107] Sainan, L., Caifeng, L., and Liming, G., 2008, A Storage Method for XML Document based on Relational Database. In *Proc. of International Symposium on computer Science and Computational Technology,* Shanghai, China, pp. 50-53.

[108] Sakr, S., 2008. Improving the Relational Evaluation of XML Queries by Means of Path Summaries. In *Proc. of the Intelligent Data Engineering and Automated Learning – IDEAL'08,* Daejeon, South Korea, LNCS 5326, pp. 378-386.

[109] Salton, G., and McGill, M.J., 1983. *Introduction to Modern Information Retrieval*. New York, NY, USA: McGraw-Hill, pp. 16-21.

[110] Schmidt, K. and Harder, T., 2010, On the use of Query-driven XML Auto-Indexing. In *Proc. of IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, Long Beach, CA, USA, pp. 81-86.

[111] Schmidt, A., Waas, F., Kersten, M., Florescu, D., Manolescu, I., Carey, M., and Busse, R., 2001. *The XML Benchmark Project.* Technical Report INS-R0103.

[112] Schoning, H., 2001. Tamino – a DBMS Designed for XML. In *Proc. of ICDE*, pp. 149-154.

[113] Shalem, M. and Bar-Yossef, Z., 2008. The Space Complexity of Processing XML Twig Queries over Indexed Documents. In *Proc. of International Conference on Data Engineering,* Cancun, pp. 824-832.

[114] Shanmugasundaram, J., Shekita, E., Kiernan, J., Krishnamurthy, R., Viglas, E., Naughton, J., and Tatarinov, I., 2001. A General Technique or Querying XML Documents Using a Relational Database System. *ACM SIGMOD Record*, 30(3), 20-26.

[115] Shanmugasundaram, J., Tufte, K., and He, G., 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *proc. of VLDB*, pp. 302-314.

[116] Silberstein, A., He, H., Yi, K., and Yang, J., 2005. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proc. of the 21st International Conference on Data Engineering, ICDE'05,* Tokyo, Japan. pp. 285-296.

[117] Sturtz Electronic Publishing (STEP). 1998. *Introduction to XML* [White paper]. Retrieved January 22, 2009, from http://www.xml.org/xml/step_intro_to_XML.shtml

[118] Suei, P., Wu, J., Lu, Y., Lee, D., Chou, S., and Lin, C., 2009. A Novel Query Pre-processing Technique for Efficient Access to XML-Relational Databases. In *Proc. of the 1st International Workshop on Database Technology and Applications*, pp. 565-569.

[119] Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E., and Zhang. C., 2002. Storing and Querying Ordered XML Using a Relational Database System. *In Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, pp. 204-215.

[120] The DBLP Computer Science Bibliography. 2009. *DBLP XML records* [Data file]. Retrieved January 22, 2009, from http://www.informatik.uni-trier.de/~ley/db/.

[121] Thompson, H., Beech, D., Maloney, M., and Mendelsohn, N. (Eds.). 2004. *XML Schema Part 1: Structures Second Edition.* Retrieved January 26, 2009, from http://www.w3.org/TR/xmlschema-1/.

[122] University of Washington. 2001. *The Tukwila system.* Retrieved January 14, 2009, from http://xml.coverpages.org/tukwila.html.

[123] Vagena, Z., Moro, M., and Tsotras, V., 2004. Twig Query Processing over Graph-Structured XML Data. In *Proc. of the 7th International workshop on the Web and Databases (WebDB'04),* Paris, France, pp. 43-48.

[124] Vakali, A., Catania, B., and Maddalena, A., 2005. XML Data Stores: Emerging Practices. *Internet Computing, IEEE*. 9(2), 62-69.

[125] Valduriez, P., 1987. Join Indices. *ACM Transactions on Database Systems (TODS),* 12(2), 218-246.

[126] Varlamis, I. and Vazirgiammis, M., 2001. Bridging XML-Schema and Relational Databases. A System for Generating and Manipulating Relational Databases Using Valid XML Documents. In *Proc. of the ACM Symposium on Document Engineering*, Georgia, USA, pp. 105-114.

[127] Vianu, V., 2003. A Web Odyssey: from Codd to XML. *ACM SIGMOD Record*, 32(2), pp. 68-77.

[128] Wang, G., Liu, M., Xu Yu, J., Sun, B., Yu, G., Lv, J., and Lu, H., 2003. Effective Schema-based XML Query Optimization Techniques. In *Proc. of 7th International Data Engineering Symposium*, Hong Kong, pp. 230-235.

[129] Wang, H., and Meng, X., 2005. On the Sequencing of Tree Structures for XML Indexing. In *Proc. of the 21st International Conference on Data Engineering, ICDE'05,* Tokyo, Japan, pp. 372-383.

[130] Wang, H., Park, S., Fan, W., and Yu., P,. 2003. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *In Proc. of the ACM SIGMOD International Conference on Management of Data,* San Diego, California, USA, pp. 110-121.

[131] Wang, W., Wang, H., Lu, H., Jiang, H., Lin, X., and Li, J., 2005. Efficient Processing of XML Path Queries Using the Disk-based F&B Index. In *Proc. of the 31st International Conference on Very Large Databases*, Trondheim, Norway, pp. 145-156.

[132] Weigel, F., 2002. A Survey of Indexing Techniques for Semi structured Documents. *Ludwig Maximilians Universitat Munchen*. Munich, Germany.

[133] Weigel, F., Meuss, H., Bry, F., and Schulz, K.U., 2004. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Advances in Information Retrieval, Proc. of 26th European Conference on Information Retrieval, ECIR'04*. Sunderland, UK, LNCS 2997, pp. 378-393.

[134] Weigel, F., Schulz, K., and Meuss, H., 2005. Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems. In *Proc. of the 7th International Workshop on Web Information and Data Management*, Bremen, Germany, pp. 23-30.

[135] Wu, X., Lee, M., and Hsu, W., 2004. A Prime Number Labeling Schemes for Dynamic Ordered XML Trees. In *Proc. of the 20th International Conference on Data Engineering,* Boston, MA, USA, pp. 66-78.

[136] Xing, G., Esanakula, J., and Jayanty, S., 2006. Index and Storage Design in Native XML Databases. In *Proc. of the 43rd Annual Southeast Regional Conference*, Kennesaw, Georgia, pp. 218-219.

[137] Xu, Y., and Papakonstantinou, Y., 2005. Efficient Keyword Search for Smallest LCAs in XML Databases. *In Proc. of the ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA, pp. 527–538.

[138] Yang, B., Fontoura, M., Shekita, M., Rajagopalan, S., and Beyer, K., 2004. Virtual Cursors for XML Joins. In *Proc. of the 13th ACM International Conference on Information and Knowledge Management, CIKM'04,* Washington, DC, USA, pp. 523-532.

[139] Yi, K., He, H., Stanoi, I., and Yang, J., 2004. Incremental Maintenance of XML Structural Indexes. In *Proc. of ACM SIGMOD International Conference on Management of Data*, Paris, France, pp. 491-502.

[140] Yoshikawa, M., Amagasa, T., Shimura, T., and Uemura, S., 2001. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases. *ACM Transaction on Internet Technology (TOIT)*, vol. 1, no. 1, pp. 110-141.

[141] Zhang, N., 2004. XML Query Processing and Optimization. *In EDBT 2004 Workshops*, LNCS 3268, pp. 121-132.

[142] Zhang, N., Kacholia, V., and Ozsu, M., 2004. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. of the 20th International Conference on Data Engineering*, Boston, USA, pp. 54-65.

[143] Zhang, C., Naughton, R., Dewitt, D., Luo, Q., and Lohman, G., 2001. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA, pp. 425-436.

[144] Zhang, N., Ozsu, M., Ilyas, I., and Aboulnaga, A., 2006. FIX: Feature-based Indexing Technique for XML Documents. In *Proc. of the International Conference of VLDB*, Seoul, Korea, pp. 259-270.

[145] Zhang, B., Wang, W., Wang, X., and Zhou, A., 2007. AB-Index: An Efficient Adaptive Index for Branching XML Queries. In *Advances in Databases: Concepts, Systems and Applications, Proc. on the 12th International Conference on Database Systems for Advanced applications, DASFAA'07,* Bangkok, Thailand, LNCS 4443, pp. 988-993.

[146] Zou, Q., Liu, S., and Chu, W., 2004. Ctree : A Compact Tree for Indexing XML Data. In *Proc. of the 6th annual ACM international workshop on Web Information and Data Management, WIDM'04,* Washington, DC, USA, pp. 39-46.

[147] Zuopeng, L., Kongfa, H., Ning, Y., and Yisheng, D., 2005. An Efficient Index structure for XML Based on Generalized Suffix tree. *Information Systems*, 32(2), pp. 283-294.

Appendix A

The datasets and queries used in testing the LLS and LTIX approaches

DBLP

The DBLP is a computer science bibliography website hosted at the University of Trier, in Germany. The DBLP server provides bibliographic information on major computer science journals and proceedings. The server initially was focused on DataBase systems and Logic Programming (DBLP). Recently, it is being expanded to include other fields of computer science. So "DBLP" now may be read as "Digital Bibliography & Library Project."

The DBLP indexes more than one million articles on computer science and contains more than 10,000 links to home pages of computer scientists. Journals tracked on this site include, to name a few, VLDB, a journal for very large databases; the IEEE Transactions; and the ACM Transactions. Proceedings are also tracked from several conferences. The DBLP XML records can be downloaded from the DBLP's website.

The DBLP XML records are used in testing the proposed LLS, LTIX, and UISX systems in this dissertation. Figure A.1 shows the number of tuples returned by the queries used to test the LLS system (Section 3.3) and the LTIX system (Section 4.3). Figure A.3 shows a small part of the hierarchical structure of the DBLP as provided by the DB2 XML Document Viewer Utilities for the DBLP data used in this dissertation.

XMark

XMark is a well-known benchmark for XML data management. It consists of a scalable document database modeling an internet auction website. XMark offers a document generator that generates XML documents of different sizes according to a numeric scaling factor. The document size grows linearly with respect to the scaling factor. For instance, factor 0.01 corresponds to a document of (about) 1.16 MB and factor 0.1 corresponds to a document of (about) 11.6 MB, and so on. The benchmark is intended to help both implementers and users to compare XML database systems independent of their own specific applications.

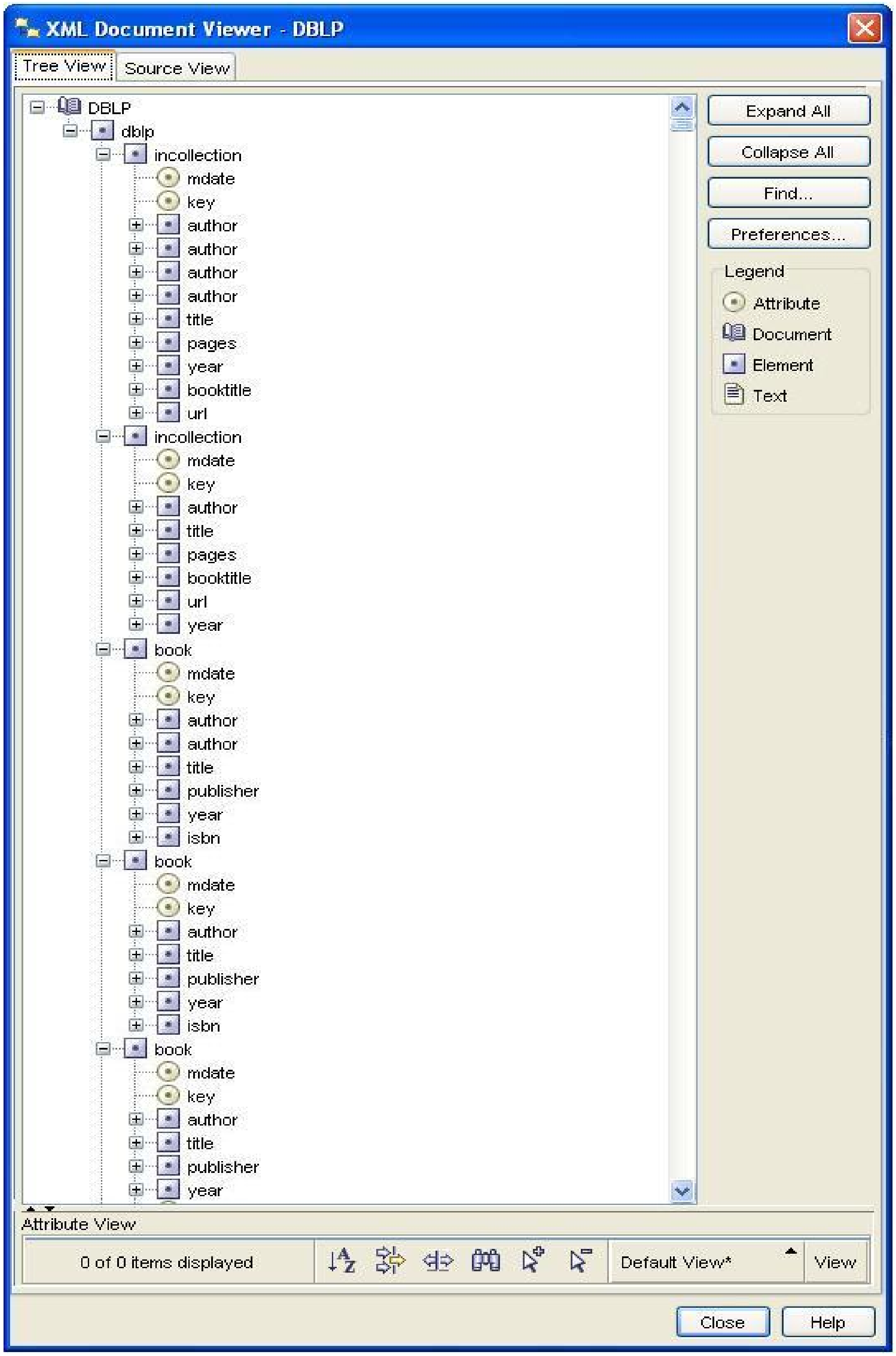
In this dissertation, we use XMark document generator to generate XML databases with different scaling factors. Then we use the generated data to test the proposed LLS, LTIX, and UISX systems. Figure A.2 shows the number of tuples returned by the queries used to test the LLS system (Section 3.3) and the LTIX system (Section 4.3). Figure A.4 shows a small part of the hierarchical structure of the XMark database used in this dissertation as provided by the DB2 XML Document Viewer Utilities.

Query No.	Query	Matching Nodes IDs	Cardin-ality	Returned Tuples
T1-Q1	/dblp/inproceedings/cdrom	<3.54>	211	211
T1-Q2	/dblp/inproceedings/cite/label	<4.7>	340	340
T1-Q3	/dblp/inproceedings/booktitle	<3.50>	43524	43524
T1-Q4	/dblp/book/series/href	<4.3>	579	579
T2-Q1	/dblp//author	see below	116276	116276
		<3.3>	5026	5026
		<3.19>	1649	1649
		<3.53>	7	7
		<3.45>	109594	109594
T2-Q2	//series/href	see below	1089	1089
		<4.3>	579	579
		<4.6>	510	510
T2-Q3	//book//label	<4.1>	2977	2977
T2-Q4	//href	see below	1238	1238
		<4.4>	59	59
		<4.2>	90	90
		<4.3>	579	579
		<4.6>	510	510
T3-Q1	/dblp/incollection[/year='2000']	<3.6>	2526	53
	/booktitle	<3.7>	2526	53
T3-Q2	/dblp/proceedings[/booktitle='ACCV']	<3.36>	794	3
	/isbn	<3.40>	733	3
T3-Q3	/dblp/inproceedings[/author='Adele E. Howe']	<3.45>	109594	12
	/title	<3.46>	43524	12
T3-Q4	/dblp/proceedings[/isbn='0-7695-1991-1']	<3.40>	733	1
	/title	<3.36>	794	1
T4-Q1	//inproceedings[/mdate='2002-08-04']	<3.43>	43524	213
	/title	<3.46>	43524	213
T4-Q2	//proceedings[/booktitle='ACNS']	<3.36>	794	5
	/isbn	<3.40>	733	5
T4-Q3	//incollection[/booktitle='Temporal Databases']	<3.7>	2526	23
	/year	<3.6>	2526	23
T4-Q4	//incollection[/author='Jurgen Annevelink']	<3.3>	5026	3
	/title	<3.4>	2526	3

Figure A.1 Number of returned tuples by the DBLP test queries

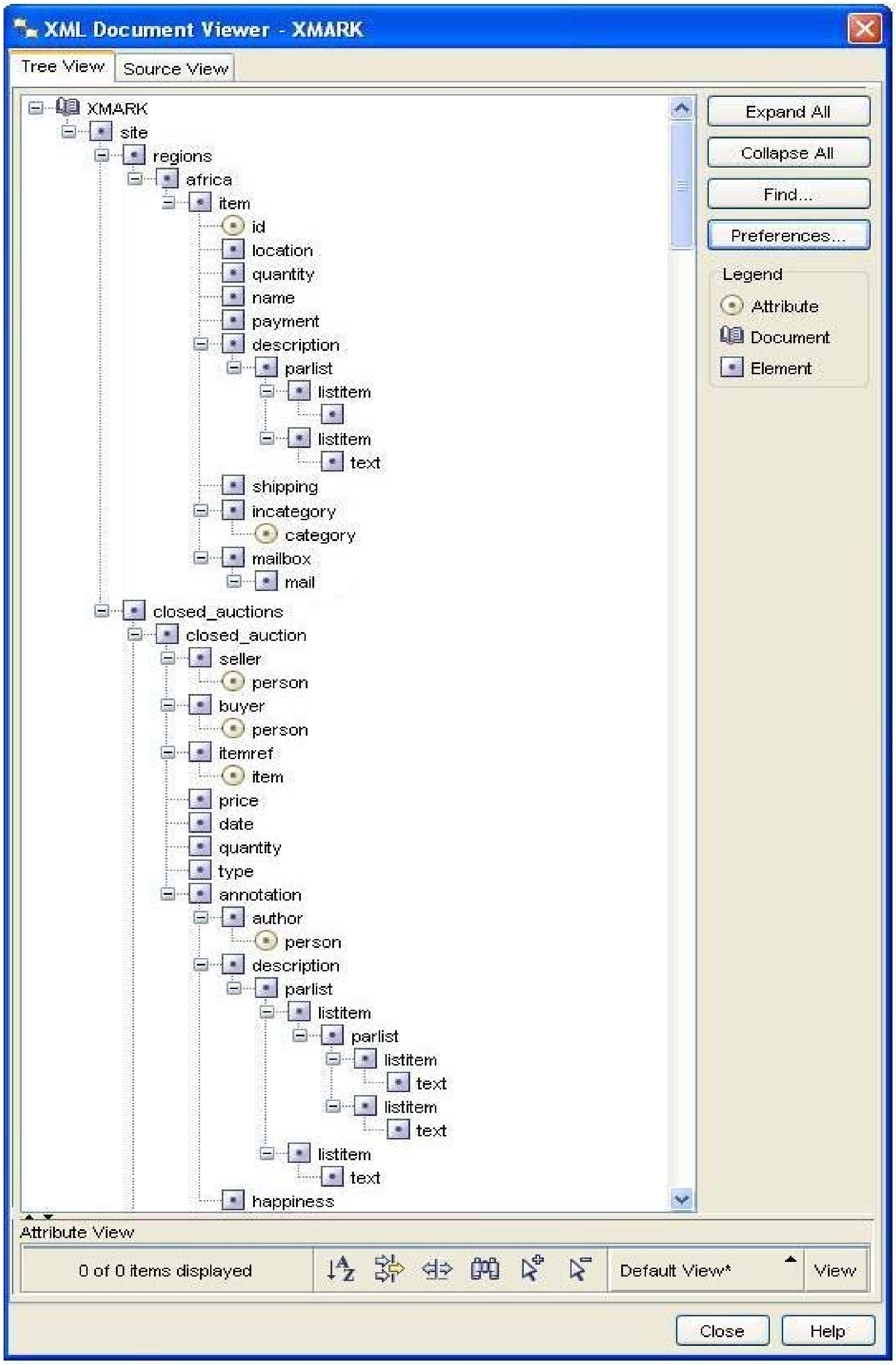
Query No.	Query	Matching Nodes IDs	Cardin-ality	Returned Tuples
T1 - Q1	/site/regions/africa/item/id	<5.1>	55	55
T1 - Q2	/site/open_auctions/open_auction/bidder/personref/person	<6.28>	6182	6182
T1 - Q3	/site/open_auctions/open_auction/seller/person	<5.80>	1200	1200
T1 - Q4	/site/catgraph/edge/from	<4.10>	100	100
T2 - Q1	//id	see below	6025	6025
		<4.7>	100	100
		<4.12>	2550	2550
		<4.21>	1200	1200
		<5.1>	55	55
		<5.11>	200	200
		<5.21>	220	220
		<5.31>	600	600
		<5.41>	1000	1000
		<5.51>	100	100
T2 - Q2	//africa//category	<6.2>	198	198
T2 - Q3	//regions//item//text	see below	6242	6242
		<8.1>	39	39
		<7.5>	53	53
		<6.4>	37	37
		<10.1>	63	63
		<8.3>	143	143
		<6.7>	137	137
		<7.10>	210	210
		<10.2>	86	86
		<8.5>	175	175
		<7.15>	212	212
		<6.12>	148	148
		<10.3>	129	129
		<6.13>	441	441
		<7.19>	590	590
		<8.7>	360	360
		<10.4>	276	276
		<6.17>	707	707
		<7.24>	985	985
		<8.9>	689	689
		<10.5>	491	491
		<6.21>	72	72
		<7.29>	88	88
		<8.11>	70	70
		<10.6>	41	41
T2 - Q4	//open_auctions//text	see below	2327	2327
		<10.7>	622	622
		<8.15>	887	887
		<6.31>	818	818
T3 - Q1	/site/regions/africa/item[/location='United States']	<5.2>	55	47
	/payment	<5.5>	53	45
T3 - Q2	/site/regions/africa/item[/id='item0']	<5.1>	55	1
	/location	<5.2>	55	1
T3 - Q3	/site/catgraph/edge[/from='category0']	<4.10>	100	1
	/to	<4.11>	100	1
T3 - Q4	/site/people/person[/name='Kaj Carey']	<4.13>	2550	1
	/phone	<4.18>	1263	1
T4 - Q1	//africa/item[/quantity='1']	<5.3>	55	52
	/name	<5.4>	55	52
T4 - Q2	//open_auction[/reserve='3199.90']	<4.23>	607	1
	/initial	<4.22>	1200	1
T4 - Q3	//closed_auction[/type='Regular']	<4.39>	975	456
	/price	<4.36>	975	456
T4 - Q4	//regions//item[/quantity='2']	see below	2175	162
	/name	see below	2175	162
	... /quantity='2'	<5.3>	55	3
	/name	<5.4>	55	3
	... /quantity='2'	<5.13>	200	17
	/name	<5.14>	200	17
	... /quantity='2'	<5.23>	220	22
	/name	<5.24>	220	22
	... /quantity='2'	<5.33>	600	36
	/name	<5.34>	600	36
	... /quantity='2'	<5.43>	1000	77
	/name	<5.44>	1000	77
	... /quantity='2'	<5.53>	100	7
	/name	<5.54>	100	7

Figure A.2 Number of returned tuples by the XMark test queries



```
DBLP
  dblp
    incollection
      mdate
      key
      author
      author
      author
      author
      title
      pages
      year
      booktitle
      url
    incollection
      mdate
      key
      author
      title
      pages
      booktitle
      url
      year
    book
      mdate
      key
      author
      author
      title
      publisher
      year
      isbn
    book
      mdate
      key
      author
      title
      publisher
      year
      isbn
    book
      mdate
      key
      author
      title
      publisher
      year
```

Figure A.3 A part of the DBLP hierarchical structure

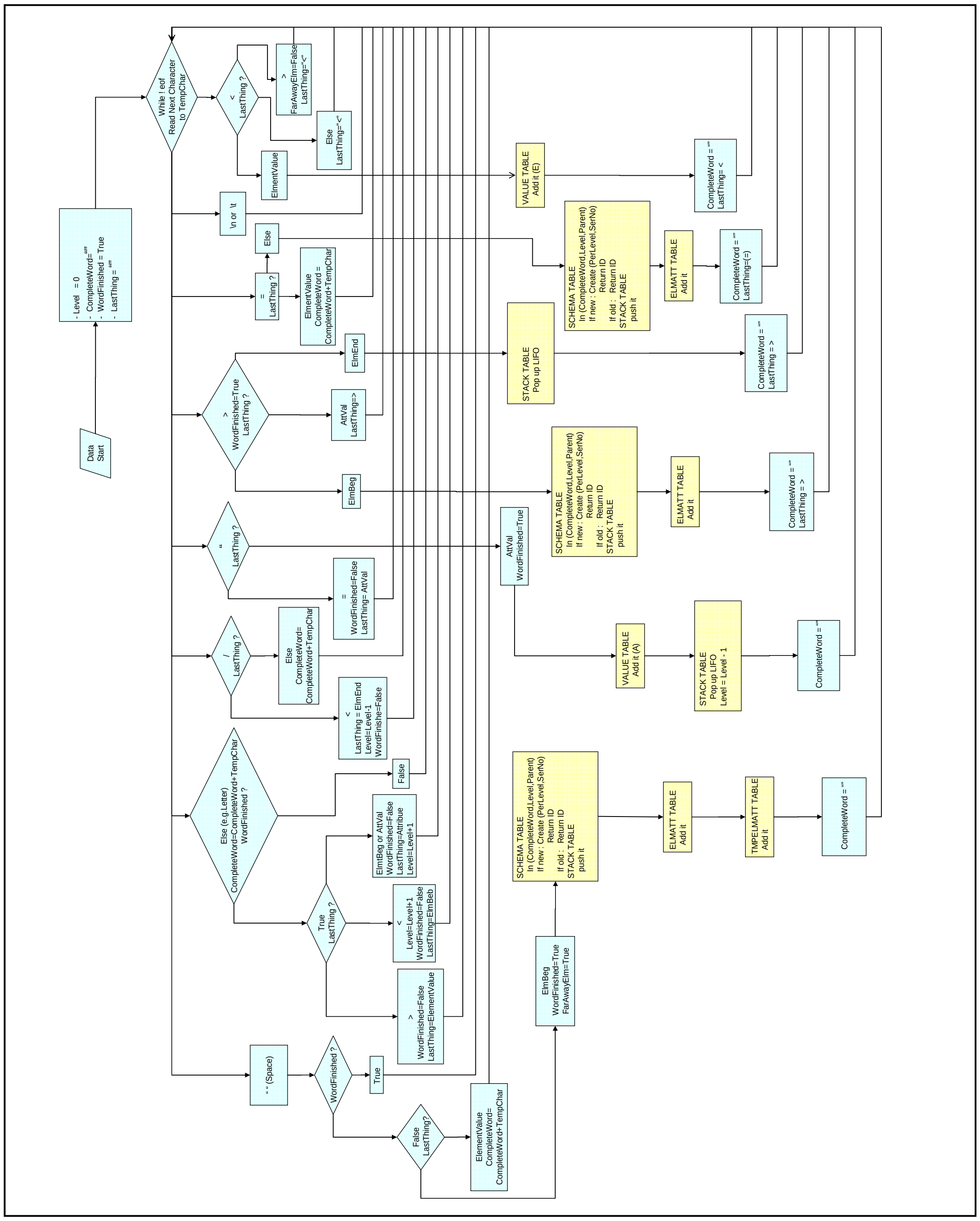


```
XMARK
  site
    regions
      africa
        item
          id
          location
          quantity
          name
          payment
          description
            parlist
              listitem
                
              listitem
                text
          shipping
          incategory
            category
          mailbox
            mail
    closed_auctions
      closed_auction
        seller
          person
        buyer
          person
        itemref
          item
        price
        date
        quantity
        type
        annotation
          author
            person
          description
            parlist
              listitem
                parlist
                  listitem
                    text
                  listitem
                    text
              listitem
                text
          happiness
```

Figure A.4 A part of the XMark hierarchical structure

Appendix B

The LLS Scanner Flowchart


Data Start
- Level = 0
- CompleteWord=""
- WordFinished = True
- LastThing = ""
While ! eof Read Next Character to TempChar
< LastThing ?
> FarAwayElm=False LastThing="<"
Else LastThing="<"
ElmentValue
VALUE TABLE Add it (E)
CompleteWord = "" LastThing= <
\n or \t
Else
= LastThing ?
ElmentValue CompleteWord = CompleteWord+TempChar
SCHEMA TABLE In (CompleteWord,Level,Parent) If new : Create (PerLevel,SerNo) Return ID If old : Return ID STACK TABLE push it
ELMATT TABLE Add it
CompleteWord = "" LastThing=(=)
ElmEnd
STACK TABLE Pop up LIFO
CompleteWord = "" LastThing = >
> WordFinished=True LastThing ?
AttVal LastThing=>
ElmBeg
SCHEMA TABLE In (CompleteWord,Level,Parent) If new : Create (PerLevel,SerNo) Return ID If old : Return ID STACK TABLE push it
ELMATT TABLE Add it
CompleteWord = "" LastThing = >
" LastThing ?
AttVal WordFinshed=True
= WordFinished=False LastThing= AttVal
/ LastThing ?
Else CompleteWord= CompleteWord+TempChar
< LastThing = ElmEnd Level=Level-1 WordFinishe=False
VALUE TABLE Add it (A)
STACK TABLE Pop up LIFO Level = Level - 1
CompleteWord = ""
Else (e.g Letter) CompleteWord=CompleteWord+TempChar WordFinished ?
False
ElmBeg or AttVal WordFinished=False LastThing=Attribue Level=Level+1
True LastThing ?
< Level=Level+1 WordFinished=False LastThing=ElmBeb
> WordFinished=False LastThing=ElementValue
SCHEMA TABLE In (CompleteWord,Level,Parent) If new : Create (PerLevel,SerNo) Return ID If old : Return ID STACK TABLE push it
ELMATT TABLE Add it
TMPELMATT TABLE Add it
CompleteWord = ""
ElmBeg WordFinished=True FarAwayElm=True
" " (Space)
WordFinished ?
True
False LastThing?
ElementValue CompleteWord= CompleteWord+TempChar