XML Structural Indexes

Samir Mohammad
Patrick Martin

School of Computing
Queen's University
Kingston, Ontario, Canada K7L3N6
{samir,martin}@cs.queensu.ca

Technical report No. 2009-560

June 2009

ABSTRACT

Extensible Markup Language (XML), which provides a flexible way to define semistructured data, is a de facto standard for information exchange in the World Wide Web. XML employs a tree-structured data model. Therefore, an XML query typically consists of two parts: structure constraints and a value predicate. Furthermore, an XML query may be either a simple single-path query with or without a predicate, or a complex twig (branching) query with or without a predicate. Indexing plays a key role in improving the execution of a query. In this chapter we give a brief history of the creation and the development of the XML data model. Then we discuss the three main categories of indexes 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.

Report Index

1. INTRODUCTION

XML is becoming the dominant method of exchanging data over the Internet. It was endorsed as a W3C recommendation in 1998 (Bary, Paoli, & Sperberg-McQueen, 1998). Its roots go back to SGML (Standard Generalized Markup Language) (Bary et al., 1998). 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 information. 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 (Sturtz Electronic Publishing [STEP], 1998). 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.

XML poses a nested hierarchical nature. An example XML document is illustrated in Figure 1. It is based on DBLP (The DBLP Computer Science Bibliography, 2009), a popular computer science bibliography dataset. We use small sets of sample data in this chapter in order to keep our examples simple. The data-tree shape in Figure 2 represents the data in the XML document of Figure 1.

A DTD is used to specify some restrictions on XML data such as, among other things, the relationship between elements and types of elements (Bary et al., 1998). XML Schema (Thompson, Beech, Maloney, & Mendelsohn, 2004) is an extension to DTD and has been supplied with many features to overcome some of the limitations of DTDs (Carey, 2004). Both DTD and XML Schema are analogous to a schema in relational Database Management Systems (DBMS). Even with the presence of a DTD and/or an XML Schema, XML data is considered as semistructured. 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 (Carey, 2004; Kaushik, Shenoy, Bohannon, & Gudes, 2002; Thompson et al., 2004).

There are many advantages to the XML data model compared with traditional data models like the relational model (Gou & Chirkova, 2007; Boag, Chamberlin, Fernandez, Florescu, Robie, & Simeon, 2007). The structure is integrated with the data in an XML document, whereas, the relational model relationships are represented by foreign keys. 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. Finally, the XML data model is flexible for querying data. This kind of flexibility does not exist in SQL (Structure Query Language) (Abiteboul, 1997).

Nevertheless, these advantages come with a cost. Since the repetition of data is irregular due to missing and/or repeated arbitrary elements, as explained above, its storage structure can be scattered over many different locations on the disk, which decreases the performance of XML queries (Chung, Min, & Shim, 2002). Furthermore, the flexibility of specifications of the XML queries (e.g. use of wild cards) adds to the challenge of indexing methods (Wang, Park, Fan, & Yu, 2003; Zou, Liu, & Chu, 2004). Also, the fact that XML documents contain the data mixed with the structure imposes a huge challenge in navigating the structural relationships among XML element sets (Jiang, Lu, Wang & Chin Ooi, 2003).

Since the creation of the XML Standards in 1998 (Bary et al., 1998), much research has been carried out to deal with these challenges. Some of them, as in the XML Lore data model (Goldman, McHugh, & Widom, 1999), are based on semi-structured data models such as the Object Exchange Model (OEM) where data in this model can be thought of as a labeled directed graph (Abiteboul, Quass, McHugh, Widom, & Wiener, 1997; Abiteboul, 1997; McHugh, Abiteboul, Goldman, Quass, & Widom, 1997). Others, such as the work by McHugh and Widom (1999), have explored the previous work on Object-Oriented query languages and extended it to permit several, possibly interrelated, path expressions in a single query (Gardarin, Gruser, & Tank, 1996) to be considered in the optimization instead of considering only one path. While the vast majority of the approaches that have been suggested to manipulate XML data are based on the relational data model (Zhang, Naughton, Dewitt, Luo, & Lohman, 2001; Tatarinov, Viglas, Beyer, Shanmugasundaram, Shekita, & Zhang, 2002; Florescu & Kossmann, 1999), other research efforts have explored the possibility of using Informational Retrieval (IR) technology (Zhang et al., 2001; Dong & Halevy, 2007; Zou et al., 2004; Guo, Shao, Botev, & Shanmugasundaram, 2003; Xu & Papakonstantinou, 2005; Weigel, Meuss, Bry, & Schulz, 2004), inverted files (Salton & McGill, 1983), and Patricia (Practical Algorithm To Retrieve Information Coded In Alphanumeric) Tries (Cooper, Sample, Franklin, Hjaltason, & Shadmon, 2001).

It is worth mentioning here that some researchers emphasize the fact that database technology has to be integrated with IR technology in order to manage XML data most efficiently (Baeza-Yates & Consens, 2004; Mariano & Baeza-Yates, 2005; Amer-Yahia, Baeza-Yates, Consens, & Lalmas, 2007). IR technology can be used to handle the unstructured text contents of XML documents, while the database technology can be used to handle the structure part of XML document.

One of the main differences between XML data and relational data is the variety of structural relationships between various elements in XML data (Che, Aberer, & Ozsu, 2006). 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. These relationships are required to manipulate XML data efficiently, however, they add more complexity to the XML data model. As a result, they make the creation of a *universal* structural index that reflects all of these relationships efficiently quite a challenging task. In the relational approach, on the other hand, the relationships are much more limited between different elements in different tables.

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. The main contributions in this chapter are:

- We use common criteria to summarize the characteristics of the most popular indexing techniques used for XML databases.

- We classify graph indexes is a novel way. Our classification is based on the presence/degree of determinism and the bisimilarity direction(s) of indexing, which control the size of an index and its query answering power, respectively.

In the remainder of this chapter we discuss a number of approaches to XML indexing. We first review the XML data models, which are used throughout this chapter, and XPath query language, which is one of the dominant query languages for XML. We next explain the three major indexing techniques used for XML data, namely, Node index scheme, Graph index scheme, and Sequence index scheme. We conclude our chapter in section 4. Comparative evaluations of these approaches are included in the context of the discussion of each approach. We divide the comparison criteria into four basic groups:

- *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.

2. BACKGROUND

XML documents can be represented as directed graphs, which consist of vertices and edges. For example, the directed graph in Figure 2 is an instance of a graph data model that represents the XML document in Figure 1. The "mapping" of an XML document to a graph may result in an acyclic graph (e.g. Figure 2), which is tree shaped, or it may result in a cyclic graph (e.g. Figure 7). While some indexes support all graph data (cyclic and acyclic graphs), others support only the tree-shaped data (acyclic graphs). In this section, we first review the main models for semistructured documents. These models are used throughout this chapter to illustrate the forthcoming concepts in XML indexing. We then review the XPath query language, which is used in this chapter to illustrate the characteristics of XML indexes, and how they can be used to query XML databases.

2.1 Data Models

Gou and Chirkova (2007) 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.

2.1.1 Edge-Labeled Tree Data Model

Figure 2 is an example of an edge-labeled model for the XML document in Figure 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 can not be repeated under the same element. Attributes are unordered and can not be nested as in elements. The element in the fifth line in Figure 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 1. DBLP like XML document

Note that in a tree structure an element can not 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 processes of an XML query.

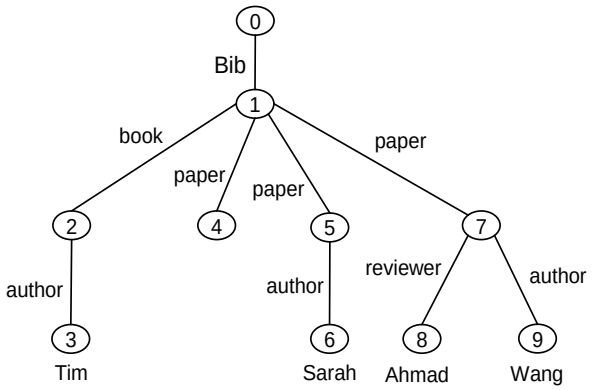


Figure 2. Edge-labeled data-tree

2.1.2 Node-Labeled Tree Data Model

Figure 3 is an example of a node-labeled data-tree for the XML document in Figure 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. For both edge-labeled and node-labeled models the hierarchal and nesting structure is self-evident in the trees that they represent.

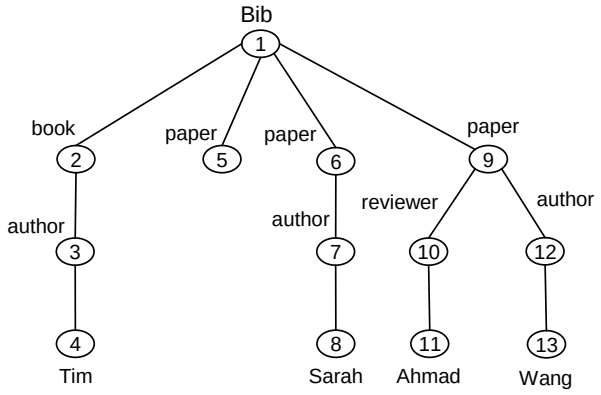


Figure 3. Node-labeled data-tree

2.1.3 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 4 is a modified version of the XML document in Figure 1. Note the use of ID/IDREF and its effect on the corresponding DAG in Figure 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 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 4. DBLP like XML document with ID/IDREF

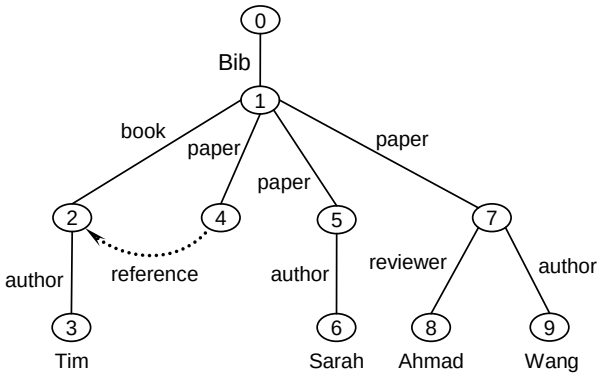


Figure 5. Directed acyclic graph data model

2.1.4 Directed Graph with Cycles Data Model

If we add an IDREF from the "book" element ("recommend=2", line number 2 in Figure 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 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 6. DBLP like XML document with ID/IDREF

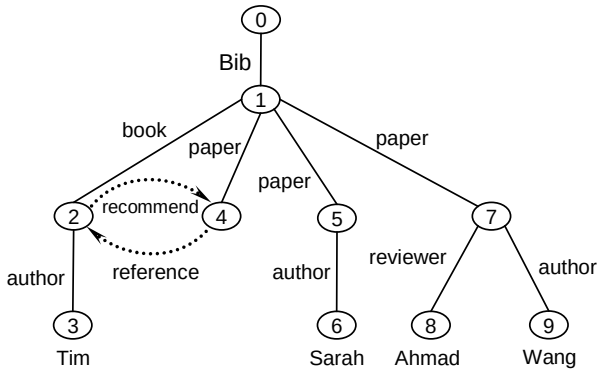


Figure 7. DBLP Directed graph with cycles data model

2.2 X-Path

Many APIs (Application Program Interfaces) have been proposed to access XML data, such as the standard Document Object Model (DOM) (Goldman, McHugh, et al., 1999) and Simple API for XML (SAX) (Megginson & Brownell, 2004). DOM XML has been defined to enable XML to be manipulated by software (Goldman, McHugh, et al., 1999). 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 (Freire & Benedikt, 2004). 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.

However, neither one of these APIs provides enough capabilities to manipulate and query XML data. Motivated by this fact, a more flexible query language, named XPath (XML Path Language) (Clark & DeRose, 1999) was proposed. Unlike other XML query languages, XPath, in addition to being able to support the main child axis "/" and the descendant axis "//", defines and supports another eleven types of axes: parent, ancestor, ancestor-or-self, descendant-or-self, following, following-sibling, preceding, preceding-sibling, attribute, self, and namespace. In this chapter we concentrate on the child "/" and descendent "//" axes.

XQuery is the other dominant language for querying XML data (Vakali, Catania, & Maddalena, 2005). Both XQuery and XPath were developed and recommended by the W3C. Furthermore, a version of XQuery (Version 1.0, 1997) is based on XPath (Boag et al., 2007).

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.

As was mentioned in the introduction, an XML query may be either a simple single-path query with or without a predicate, or a complex twig query with or without a predicate. A complex twig query with a predicate specifies patterns of selection on multiple elements related to one another by a tree structure. For example, "q1" below is a simple path query.

q1: /Bib/paper/author

If we run this query against the XML document in Figure 3 above, it returns the texts "Sarah" and "Wang" which are the values of the "author" elements under the "paper" elements under the "Bib" element. Query "q2" illustrates the use of the descendent axis.

q2: /Bib//author

This query returns {“Tim”, “Sarah”, “Wang”}, which represents all author elements under the top-level “Bib” element. Query “q3” is an example of a complex twig query.

q3: //paper[/reviewer=“Ahmad”]/author

This query asks for the author of a paper that has a reviewer “Ahmad,” and the query returns the author “Wang.” It 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 “q3” 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 8. In this figure, a circle 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. (Gou & Chirkova, 2007).

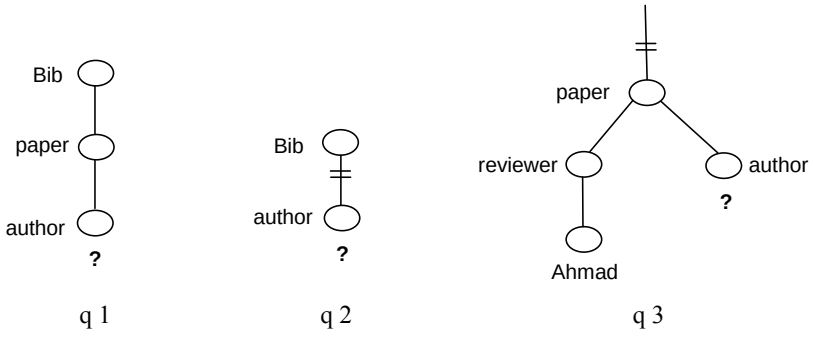


Figure 8. Schematic representation of XPath queries.

3. STRUCTURAL INDEXING SCHEMES FOR XML DATA

The structural index of an XML database is 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, in the case of XML structural indexes, some index types such as a graph index are used to determine if an XML path exists, before going any further in the query processing for both simple path queries and complex path queries (with branching elements). In this section, structural indexes for XML are analyzed in detail.

Generally, structural indexes can be grouped into three categories:

- *Node indexes* (Li & Moon, 2001; Zhang et al., 2001; Grust, 2002): These schemes depend on many labeling approaches including interval labeling (Dietz, 1982), and prefix labeling (Tatarinov et al., 2002; Online Computer Library Centre, 2008; Yang, Fontoura, Shekita, Rajagopalan, & Beyer, 2004; Lu, Ling, Chan, & Chen, 2005). Both of these labeling approaches are best suited for tree shaped data.
- *Graph indexes*: These schemes contain indexes that cover either path queries only (Polyzotis, Garofalakis, & Ioannidis, 2004; Chung et al., 2002; Cooper et al., 2001; Goldman & Widom, 1997), or both path and twig queries (Kaushik, Bohannon, Naughton, & Korth, 2002; Wang, Wang, Lu, Jiang, Lin, & Li, 2005). We divide graph indexes in this chapter into three types depending on their deterministic property and bisimilarity direction(s) (see *Graph Indexing Schemes* section).
- *Sequence indexes* (Rao& Moon, 2004; Wang, Park, et al. 2003; Wang & Meng, 2005): They interpret the whole query as structured series of sequences and search for a match in the structured encoded sequence of an XML document.

Please note that the term "path indexes" 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 chapter we prefer not to use the term "path indexes" and use the specific terms above in order to eliminate any ambiguity.

3.1 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 returned answer is dealt with in one of the following two ways: (1) We consider this answer complete, precise, and the final one; (2) We consider this answer as a primary answer and pass it to further post-processing stage(s) to double-check the precision of the initial returned answer before approving it. 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 it does not return any false answers. In other words, the returned answer does not contain any incorrect answers, among the correct answers. Precision is considered for both path queries and twig queries.
- *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. A structural index supports a complete answer if it guarantees that the returned answer for a given query contains a complete set of all possible answers. In other words, the returned answer may be a superset of the correct answer. This criterion is important, because we do not want to miss any part of the correct answer. The surplus parts of the result can be eliminated by some post-validation step. Recall is considered for both path queries and twig queries.

- *Processing complexity:* This criterion covers different kinds of complexity depending on the type of indexing scheme that is used. It covers topics such as the primary processing procedure and the additional processing cost of required joins.
- *(A) Scalability*: Large indexes may involve many 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.
  (B)- 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. 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 are path queries and twig 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 numbering scheme that is used to label nodes. We study two types of updates: (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).

3.2 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 numbers can be used to solve simple path and twig path queries. Paths are solved 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 (Al-Khalifa, Jagadish, Koudas, Patel, Srivastava, & Wu, 2002; Bruno, Koudas, & Srivastava, 2002; Chien, Vagena, Zang, Tsotras, & Zaniolo, 2002; Li & Moon, 2001; Zhang et al., 2001). Structural joins are explained in *Index for Structural Joins* section.

Labeling (numbering) schemes were used prior to the creation of XML to reproduce the structure of a tree (Dietz, 1982). Two of the most widely used types of schemes are interval (a.k.a. region) labeling and prefix (a.k.a. path) labeling. In the following, we take the (Beg, End)

labeling scheme as an example of the first type and the Dewey labeling scheme as an example of the second type.

3.2.1 Criteria for Evaluation of Node Indexes

In addition to the evaluation criteria listed in *Criteria for Evaluation of Structural Indexing Schemes* section, we refine the processing complexity criterion into the following criteria that are applicable specifically to node indexing schemes.

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 solve queries with the "//" axis.
  - *Parent/child relationship*: It is useful to solve queries with the "/" axis.
  - *Sibling relationship*: In some cases, a group of sibling nodes form an answer for a twig query. For example, finding the name of the author of the paper with reviewer = "Ahmad" in Figure 3.
- *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 solve a query.
- *Data type used in indexing scheme*: Comparing different data types involve different algorithms with different operations. As an illustration, comparing two numbers usually requires less time than comparing two sequences of strings.

3.2.2 Interval Labeling Scheme

The (Beg,End) and (Pre,Post) labeling schemes are examples of interval labeling scheme. Zhang et al. (2001) introduce the (Pre,Post) labeling scheme – which was invented by Dietz in 1982 – to index the elements in a document. They assign a pair of numbers to each node that represents the pre-order and post-order traversal number of an XML tree. The (Beg,End) labeling scheme is basically the same. 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 an attribute or an attribute value, we assign to that 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 we reach the ending tag of an element, we assign the "End" number for it (which is equal to the next available sequential number). Since the value of an element is a leaf node, the "Beg" number of this value is equal to the "End" number. Figure 9 is an example of (Beg,End) labeling scheme for

the XML document in Figure 1. The beginning and the ending numbers imply the positions of the opening tag (<..>) and the closing tag (</..>), respectively, in an XML document.

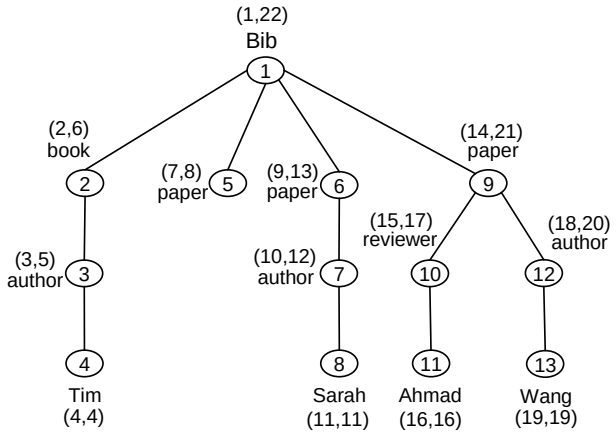


Figure 9. (Beg,End) Labeling Scheme

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 (Zhang et al., 2001). 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" iff x.Beg < y.Beg < x.End.
For example, in Figure 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" iff (x.Beg < y.Beg < x.End and y.Level = x.Level + 1.
For example, in Figure 9, node (1,22,1) is a parent to the node (2,6,2).

The interval labeling scheme (Beg,End) can be used to solve a twig query by leveraging the power of relational DBMS technology, and by using "structural joins" (shortly "joins"). Zhang et al. (2001) use inverted lists of a node-shaped data-tree to solve XML queries. They shred XML documents into relational tables with the fixed schema (Label, Beg, End, Level, Flag, Value) (Gou & Chirkova, 2007). Table 1 is an example that represents the relational table of the shredded XML document of the node-label tree in Figure 9. Instead of storing all shredded tuples in the same table, we can extend this approach to a binary approach in which shredded tuples are grouped in separate tables that are based on "Label" types.

Label (Tag)	Beg	End	Level	Flag (Type)	Value
book	2	6	2	Element	Null
author	3	5	3	Value	Tim
paper	7	8	2	Element	Null
…	…	…	…	…	…
paper	14	21	2	Element	Null
reviewer	15	17	3	Attribute	Ahmad

Table 1. A node table of the XML data in Figure 9

Figure 10 shows the SQL transformation of the query "q3" below. This transformation is based on the binary approach of the node-labeled tree data. Because this approach is based on (Beg,End) labeling scheme, it uses inequality comparisons to find the Ancestor/Descendent and the Parent/Child (containment) relationships.

q3: //paper[/reviewer="Ahmad"]/author

```
Select author.Value
From   paper,reviewer,author
Where  paper.level=2                  and
       paper.start<reviewer.start     and
       reviewer.start<paper.end       and
       paper.level+1=reviewer.level   and
       paper.start<author.start       and
       author.start<paper.end         and
       paper.level+1=author.level     and
       reviewer.Value="AHMAD"
```

Figure 10. SQL equivalence of "q3" using binary approach of node-labeled tree

The relations that are supported by the node approach are the parent/child ("/") and the ancestor/parent ("//") relationships. The (Beg,End,Level) labeling scheme, in Table 1, is used to infer the relationship between only two nodes at a time. It requires only a single comparison to infer any of these two relations; however, the number of joins required to evaluate an XML query by using the previous relational node approach is equal to the number of nodes in the query minus one, which is high for large twig queries. Furthermore, the inequality operator that is used in the join operations is designed for relational DBMSs, and is unsuitable for the joining of XML elements. Motivated by these shortcomings of the relational approach, many native approaches have been developed to query an XML document more effectively(Gou & Chirkova, 2007; Zhang et al., 2001; Al-Khalifa et al., 2002; McHugh & Widom, 1999).

The (Beg,End) labeling scheme is used to solve both path queries and twig queries. 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 can not 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 can not be determined.

Temporal XML databases 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 data source over time by reviewing historical data. The paper by Cohen, Kaplan, and Milo (2002) 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 (2001), suggest leaving gaps between the interval values for new nodes to be inserted. These durable approaches may provide intervals for a certain number of new nodes equal to the gap size. After filling these gaps, renumbering or other solutions become inevitable. Cohen et al. (2002) proved immutable (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, He, Yi, and Yang (2005) and Chen, Mihaila, Bordawekar, and Padmanabhan (2004) have designed dynamic labeling structures for interval indexes that allow relabeling by using only O(log n) bits per label.

Fortunately, 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 constant time by using a comparison operation between the index numbers. Nevertheless, updating the labeling (numbering) 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 (1982) pioneered the labeling of an ordered tree (Gou & Chirkova, 2007; Li & Moon, 2001). He used (Pre-order, Post-order) numbers to label (index) 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 preorder sequence. A vertex "x" is an ancestor of "y" iff "x" occurs before "y" in the pre-order traversal of the tree and after "y" in the post-order traversal. Li and Moon (2001) propose the (Order,Size) labeling scheme. The "Order" part is based on a preorder 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 re-labeling of the data-tree in case of insertion. Re-labeling may be delayed, but eventually it is required. It occurs more often if the data distribution in the tree is skewed.

Tatarinov et al. (2002) discuss the possibility of using real 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 any difference. Later, Amagasa, Yoshikawa, and Uemrua (2003) used real numbers instead of integers to represent a region (interval) in node indexing. Similar to the (Order,Size) labeling scheme (Li & Moon, 2001), the real numbers approach only avoids node re-labeling as much as possible. If the number of insertions exceeds a specific limit, the nodes have to be re-labeled. Wu, Lee, and Hsu (2004) 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).

3.2.3 Prefix Labeling Scheme

Dewey labeling, which is an example of a prefix labeling scheme, is another coding scheme that was originally made for general knowledge classification (Online Computer Library Centre, 2008). Tatarinov et al. (2002) introduce it to 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 labeling index since each node is represented as a complete path from the root to the indexed node.

Figure 11 is an example of the Dewey labeling scheme for the XML document in Figure 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 its 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 11) in the Dewey label.

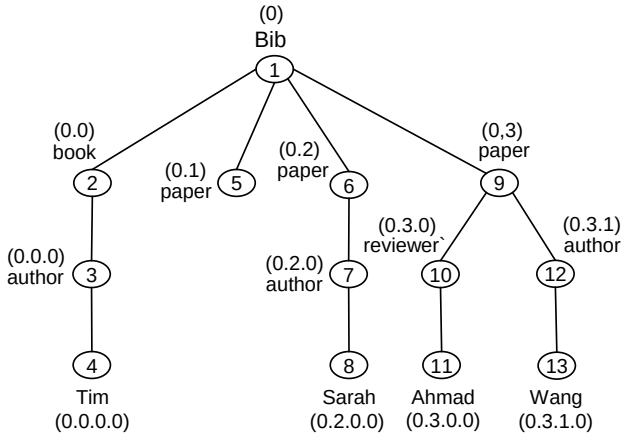


Figure 11. Dewey labeling scheme

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 label provides direct support for the sibling relationship. In a given tree, node “x” and node “y” are siblings iff 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 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 (Tatarinov et al., 2002). However, its storage size increases tremendously as the depth of the tree increases. 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, Lam, Shui, and Wong (2006) 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 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 (Cohen et al., 2002).

It is easy to infer the exact ancestor or descendent of a given node in Dewey labeling scheme indexes. For example, in Figure 11 the ancestors of the node (0.3.1) are the nodes that start with (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 solved, which is equal to the number of nodes in the query minus one.

Many variants of prefix labels are proposed in the literature. O’Neil, O’Neil, Pal, Cseri, Schaller, & Westbury (2004) 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 GRoup base Prefix (GRP) labeling scheme (Lu & Ling, 2004) the labels consist of two parts, namely, group ID and group prefix. Doung and Zhang (2005) 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 (immutable), therefore their label sizes can reach O(n) bits per label in the worst case.

3.2.4 Summary of Node Indexes

Table 2 below contains a summary of the two types of labeling schemes. The precision of an index scheme could be 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 constant if we can determine the relationship between any two arbitrary nodes in constant time, regardless of the depth of the data-tree. The relationships that are supported could be ancestor/descendent, parent/child, and sibling relationships. The data type could be either a number or a string. The types of queries that are supported by these node indexing schemes are path and twig queries. The evaluation of these queries may 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.

No.	Criteria		Interval Labeling (Beg,End)	Prefix Labeling (Dewey)
1	**Precision**		Precise	Precise
2	**Recall**		Complete	Complete
3	**Computation Complexity**	**Relationship Computation**	Constant	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 efficiently**		None	None
6	**Maintenance cost**	**Mutable**	O (log n)	O (log n)
		Immutable	O (n)	O (n)

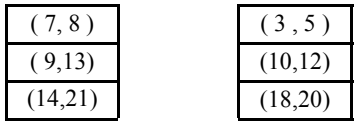
Table 2. Comparison of interval labeling scheme and prefix labeling scheme

Both types are equivalent with respect to precision, completeness (recall), and maintainability. However, they differ with respect to the other characteristics (computation complexity, and size/scalability). We notice that each type's advantages are the disadvantage of the other. The (Beg,End) labeling scheme requires constant 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 regardless of the level (depth) at which each node is located. On the contrary, 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 contains 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.

3.2.5 Indexes for Structural Joins

Using an appropriate labeling scheme to reflect the structure and the contents of XML data is known as node indexing, that is, indexing an XML tree based on its node granularity. Structural join indexes further provide an efficient access to these node indexes. Structural joins of elements in an XML data-tree that uses interval coding have received much attention in the research community (Zhang et al., 2001; Li & Moon, 2001; Al-Khalifa et al., 2002; Bruno et al., 2002; Chien et al., 2002; Jiang et al., 2003; Li, Lee, Hsu, & Chen, 2004). Structural joins are performed between the inverted lists (which are basically node indexes) of two elements to establish a parent/child or ancestor/descendent relationship. This is analogous to joining two tables in the relational approach. In XML, however, the advancing mechanisms of the join or loop cursor(s) are modified in accordance with the index values of the interval for the elements under inquiry. For example, assume that the "paper" and "author" elements in Figure 9 are stored in the node indexes shown in Figure 12. To evaluate the query "//paper/author" against these indexes, a skipping mechanism should match node (9,13) only with node (10,12) and skip nodes (3,5) and (18,20). Similarly, node (14,21) should be matched only with node (18,20) and the first two nodes in the "author" list should be skipped. Such an *intelligent* skipping mechanism reduces the number of I/O accesses and improves the query evaluation.



(A) paper (B) author

Figure 12. inverted lists of "paper" and "author" elements in Figure 9

The earliest works in structural joins using interval coding are the Multiple Predicate MerGe JoiN (MPMGJN) by Zhang et al. (2001) and XISS by Li and Moon (2001). Both of these approaches require, in the worst case, an element set to be scanned multiple times during the join operations of two elements. As an improvement, Al-Khalifa et al. (2002) propose a primitive data structure called a Stack-Tree. Their approach is based on scanning two inverted indexes completely only once in order to explore the existing relationships between the elements of these indexes. Chien et al. (2002) further improve the scanning of the indexes. To investigate the descendent relation, their approach examines only the related nodes and skips nodes that do not have a match. The ancestor skipping mechanism, however, only skips small parts of the nodes that do not have a match. Approaches that skip descendents as well as ancestors include the Holistic twig join approach (Bruno et al., 2002) and XR-stack (Jiang et al., 2003).

The query evaluation plan that uses structural joins is based on sets of nodes, which are merged together to infer a structure. Thus, it is referred to in the literature as a set-based query process (Jiang et al., 2003; Moro, Vagena, & Tsotras, 2005).

3.3 Graph Indexing Schemes

A Graph index (a.k.a. Summary Index) is a structural path summary 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. Examples of graph indexes are DataGuides (Goldman & Widom, 1997; Goldman & Widom, 1999), Index Fabric (Cooper et al., 2001), APEX (Chung et al., 2002), D(k)-index (Chen, Lim, & Ong, 2003), $(F+B)^k$-index (Kaushik, Bohannon, Naughton, & Korth, 2002), and F&B-index (Abiteboul, Buneman, & Suciu, 2002; Gou & Chirkova, 2007).

Graph indexes consider paths, during query evaluation, as a whole path instead of dealing with each node in the path separately (such as the node indexing scheme). A subsequent step is needed to join simple paths together in order to solve a twig query. In contrast to the node scheme, 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 (2007) classify them into two classes, path indexes (P-indexes), which are able to cover simple path queries (such as DataGuides and 1-index), and twig indexes (T-indexes), which are able to cover twig queries (such as F&B-index). Graph indexes can also be categorized according to their path exactness (Polyzotis & Garofalakis, 2002). Some schemes are exact such as strong Data Guide, Index Fabric, 1-index, F&B-index, and disk-based F&B-index, while others are approximate such as approximate Data Guide, A(k)-index, D(k)-index, and $(F+B)^k$-index.

Our classification, in this chapter, is based on path determinism and bisimilarity.

Path determinism: If the index tree is a Deterministic Finite Automata, then the paths of the tree are considered to be deterministic paths. 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.

Bisimilarity: There are two types of bisimilarity, namely, forward and backward bisimilarity. Two nodes are backward bisimilar if they share the same incoming paths. Similarly, two nodes are forward bisimilar if they share the same outgoing paths. Partitioning of all elements in a data-tree based on their forward and backward bisimilarity is much better than having them partitioned based only on their backward bisimilarity, because forward and backward bisimilarity provides efficient and precise support for twig queries.

Based on the path determinism and the bisimilarity, we classify graph indexes as follows:

- *Deterministic graph indexes:* This includes DataGuides (Goldman & Widom,1997), approximate Data Guide (Goldman & Widom, 1999), and Index Fabrics (Cooper et al., 2001).
- *Non-deterministic graph indexes with backward bisimilarity:* This includes 1-index (Milo & Suciu, 1999), A(k)-index (Kaushik, Shenoy, et al., 2002), and D(k)-index (Chen, Lim, et al., 2003).
- *Non-deterministic graph indexes with forward and backward bisimilarity:* This includes F&B-index (Gou & Chirkova, 2007; Wang, Wang, et al., 2005; Abiteboul, Buneman, et al., 2002), $(F+B)^k$-index (Kaushik, Bohannon, Naughton, & Korth, 2002), and disk-based F&B-index (Wang, Wang, et al., 2005).

Gou and Chirkova (2007) classification combines our first two groups into one that covers simple 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 simple path queries (where the complete path is known). For example, to evaluate the query "/P/A" over the deterministic strong Data Guide index in Figure 13(B) we have to traverse one path only. In contrast, non-deterministic graph indexes may lead to traversing more than one index path to solve a simple path query. For example, to evaluate the same query as described above over the non-deterministic 1-index in Figure 13(C) we have to traverse more than one path that satisfies the query. On the contrary, non-deterministic graph indexes 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 Data Guide index in Figure 13(B) is listed twice, while the non-deterministic 1-index in Figure 13(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 (Milo & Suciu, 1999). 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 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.

In the following sections, we elaborate the development of graph index schemes of these three classes. The evaluation criteria listed in *Criteria for Evaluation of Structural Indexing Schemes* section will be used to analyze each indexing scheme. Please note that all graph indexing schemes provide a complete answer for both path queries and twig queries. They do not require extra joins to evaluate the path queries but they require join operations to solve the twig queries.

3.3.1 Deterministic Graph Indexes

In deterministic graph indexes, each unique path in a data-graph is listed only 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 are reviewed in this section, namely, strong DataGuides, approximate DataGuides, and Index Fabrics.

3.3.1.1 Strong Data Guide

Goldman and Widom (1997) presented one of the early structure summaries called a strong Data Guide. 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.

An example of the strong Data Guide is shown in Figure 13(B), which represents the summary of the data in Figure 13(A). To simplify the comparison between different schemes in Figure 13, 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 13(B) stand for book, paper, author, and reviewer in Figure 13(A), respectively. Figure 13 (A) is a modified version of the edge-labeled data-tree in Figure 2. The difference is that two edges are inserted (represented by the dashed lines in Figure 13 (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 into directed acyclic graph-shaped data. Unlike node indexes, graph indexes are capable of supporting the directed acyclic graph data model.

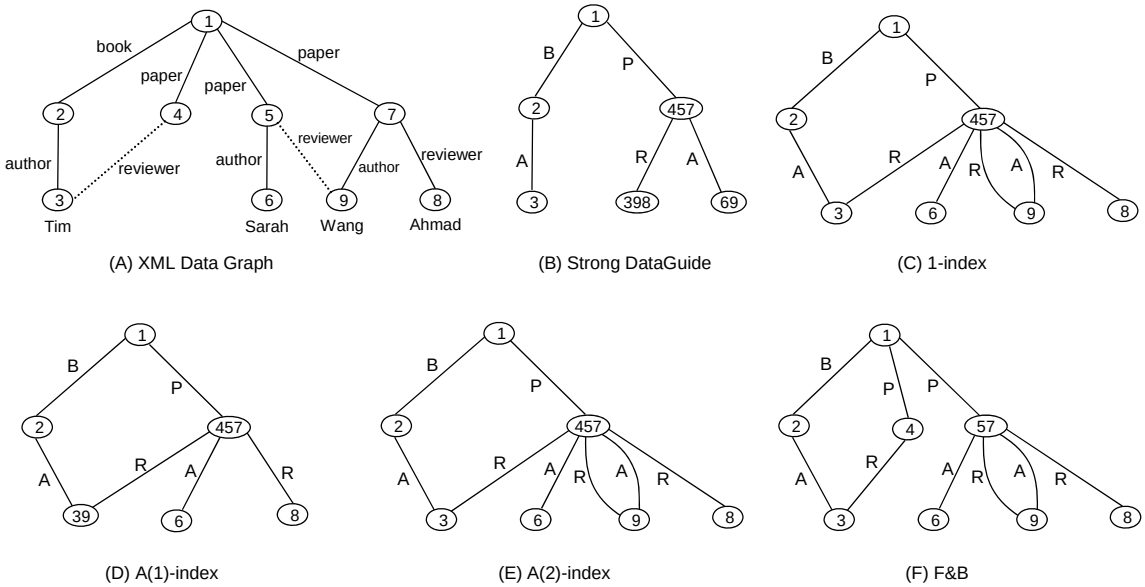


Figure 13. XML data-tree and its corresponding graph indexes

The graph index (a.k.a. structure summary) of an XML data-graph is a strong Data Guide 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.

The graph index in Figure 13(B) is a strong Data Guide for the data in Figure 13(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 Data Guide 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. The case of 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 simple parent/child path queries (Kaushik, Bohannon, Naughton, & Korth, 2002) 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 (Kaushik, Bohannon, Naughton, & Korth, 2002). For example, evaluating "/P[/A]/R" query – which returns an "R" node that has a "P" parent node and an "A" sibling node – over the strong Data Guide index in Figure 13(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 Data Guide. In the worst case, updating a tree with a subgraph of structured data that has loops and sharing may incur recomputation to a large portion of the strong Data Guide. Both types of updates, namely, edge and subgraph additions are supported by the strong Data Guide scheme. 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 Data Guide.

3.3.1.2 Approximate Data Guide

Experiments have shown, in general, that the strong Data Guide size is much smaller than the original database. There are cases, however, where the size of the strong Data Guide is unreasonably large (e.g., for cyclic data). To overcome this disadvantage, an Approximate Data Guide (ADG) is proposed by Goldman & Widom (1999). ADG ignores the second requirement of the strong Data Guide, 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 Data Guide characteristics are applicable for ADG, except that the size of the ADG is often smaller.

3.3.1.3 Index Fabric

Index Fabric was proposed by Cooper et al. (2001) 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 which is based on Patricia tries (Knuth, 1998). 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 many 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 illustration, each edge of the data-tree in Figure 14(A) (which is the same as the XML data-tree in Figure 2) is given a designator as illustrated in Figure 14 (B). The edge labels along with the content of the data-tree are combined at the leaf nodes to form a path index for each value in the tree. Note that compression is used to minimize the size of the tree as follows. In Figure 14(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.

Note that the Index Fabric in Figure 14 (C) is similar to the strong Data Guide in Figure 13 (B). Index Fabric is conceptually similar to strong Data Guide (Wang, Park, et al., 2003; Chung et al., 2002; Weigel et al., 2004), so 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 when it is used to index tree-shaped data.

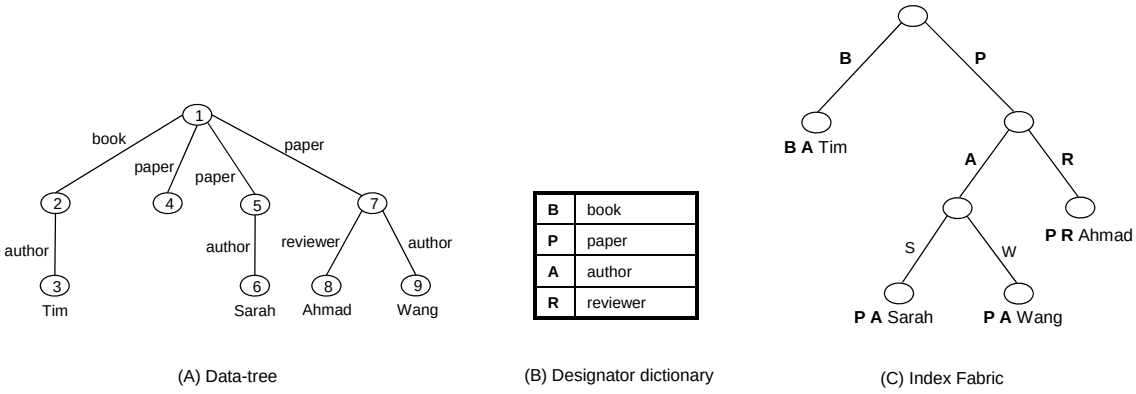


Figure 14. Index Fabric of the data-tree in Figure 2

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. The update algorithm for subgraph addition to the Index Fabric is not published to the best of our knowledge.

3.3.2 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.

3.3.2.1 (1-index)

Milo and Suciu (1999) propose 1-index as an attempt to reduce the size of a structural summary to less than that of a strong Data Guide by relaxing the determinism constraint. Figure 13(C) is an example of 1-index for the data in Figure 13(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 Data Guide and 1-index are identical in the case of simple 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 13(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 and irregular (Chen, Lim, et al., 2003).

While a 1-index represents every value in the source data only once in the index tree, a strong Data Guide 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 Data Guide represents all similar paths in the source data by only one path in the index. Therefore, strong Data Guide is more *path centric* in its partition.

It is easy to see from Figure 13(C) that 1-index is complete and precise for evaluating path queries such as "/B/A" and "//R", and it is complete, but not precise for evaluating twig queries like "/P[/A]/R". In General, 1-index is always complete, but not necessarily precise (Kaushik, Shenoy, et al., 2002).

Kaushik, Bohannon, Naughton, and Shenoy (2002) reviewed 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*, *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 update algorithm for edge addition is called *propagate*. The complexity of the algorithm for edge addition is measured by two factors. First, by the difference between the refined propagated index and the original 1-index. This difference can be as large as O(n) in the worst case, where n is the number of nodes in G. Second, the complexity is measured by the number of nodes and edges touched in the data-graph during propagation, which can be *O(n + m)* in the worst case scenario, where *m* is the number of edges in *G* (Kaushik, Bohannon, Naughton, & Shenoy, 2002).

3.3.2.2 A(k)-index

The dominant disadvantage of strong Data Guide and 1-index is the size of their indexes when the source data is large and irregular. A(k)-index is proposed by Kaushik, Shenoy, et al. (2002), mainly to overcome the size problem. Similar to 1-index, A(k)-index (Figure 13 (D & E)) is based on backward bisimilarity. A(k)-index is also a non-deterministic node centric index. 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 are generally smaller than that of a strong Data Guide 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 (Kaushik, Shenoy, et al., 2002). Let us take an A(1)-index for the data in Figure 13(A), which is illustrated in Figure 13(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". A(k)-index partitioning is based on backward bisimilarity. 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 13(E), is complete and precise for both "/B/A" and "//R" queries. Note that Figure 13(E) is identical to the 1-index in Figure 13(C). Actually, a 1-index is a special case of A(k)-index where "*k*" value is equal to the depth of a data-tree (the longest path in a tree). A(k)-index is complete but not precise for twig queries like "/P[/A]/R".

The subgraph addition algorithm of 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 (Kaushik, Bohannon, Naughton, & Shenoy, 2002).

3.3.2.3 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, Lim, et al. (2003) 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, Bohannon, Naughton, and Shenoy (2002). On the other hand, the edge addition algorithm is novel and 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* (Chen, Lim, et al., 2003).

3.3.3 Non-deterministic Graph Indexes with (Forward & 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 simple path queries.

3.3.3.1 F&B-index

The F&B-index was introduced by Abiteboul, Buneman, et al. (2002). 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 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", which returns the "R" nodes that are children of "P" nodes and siblings of "A" nodes. Evaluating this query over strong Data Guide (Figure 13(B)), 1-index (Figure 13(C)), or A(2)-index (Figure 13(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 previous graph indexes. Eventually, it is eliminated from the final answer after performing some additional join steps. In contrast, as illustrated in Figure 13(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). However, insufficient memory problems may arise for very large size indexes. Kaushik, Bohannon, Naughton, and Korth (2002) 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, Bohannon, Naughton, and Shenoy (2002) can be adopted.

Kaushik, Bohannon, Naughton, and Korth (2002) 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*" (Gou & Chirkova, 2007). 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.

3.3.3.2 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, Wang, et al. (2005) 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 (Wang, Wang, et al., 2005) did not discuss or present any updating algorithm for their indexing scheme.

3.3.4 Summary of Graph Indexes

Note that 1-index and strong Data Guide 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 simple 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 3 contains a summary of the graph indexing schemes. The precision of an index scheme could be 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. 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 (the third type) are the only type of graph indexes that are capable of supporting twig queries if the index is exact (i.e. F&B-index or disk-based 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.

		Deterministic	Non-deterministic Backward Bisimilar	Non-deterministic Forward & Backward Bisimilar
Criteria		**strong DataGuide, Approximate DataGuide, Index Fabric**	**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	Yes
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>		Path	Path	Path (Twig by F&B-index and disk-based F&B-index)
6- Maintain ability (Edge insertion)		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)	- Path & twig queries precision depends on "*k*" value for $(F+B)^k$-index - Maintainability of disk-based is not available (open for research)

Table 3. Comparison between the three categories of graph indexing approaches

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. For example, it can hold sample values for each node or statistics about the extended data such as fan-in and fan-out of each node.

3.4 Sequence Indexing Schemes

Sequence indexes (Wang, Park, et al., 2003; Rao & Moon, 2004) 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. However, we should be careful when a query is answered by matching the sequences, since the sequence may not necessarily reflect a structural tree match (see "Refinement step" in *Specific Comparison Criteria of Sequence Indexes* section below). Sequence indexes combine the structure and the values of XML data into an integrated index structure. They are used to efficiently evaluate path queries as well as twig queries with keyword components without any extra join operations with some tables that may hold the values.

3.4.1 Specific Comparison Criteria of Sequence Indexes

In addition to the comparison criteria listed in section 3.1, we include the following comparison criteria for this type of indexes.

- *Computation 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 several hundreds, thousands, or more leaf nodes. Leaf nodes are usually value nodes. We have two ways to map XML elements in a tree, namely, in a top-down direction or a bottom-up direction. A top-down search for a value in a data-tree has to start from the root element then go 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 value nodes (at the bottom) is higher than that of the element nodes in the top and the middle of the tree (i.e., the value labels at the bottom of the tree may occur less frequently than the element labels found in the higher levels), bottom-up search results in fewer paths in the tree that need to be examined. Therefore, the indexing direction has an effect on the efficiency of a query evaluation.
- *Refinement step*.
  Sequence schemes suffer from two anomalies, namely, false alarms (a.k.a. false positives or imprecise result) and false dismissals (a.k.a. false-negatives 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 divided sequence indexes into two types, namely, top-down sequence indexing schemes and bottom-up sequence indexing schemes.

ViST and PRIX are discussed in the next two sections, as examples of top-down and bottom-up types, respectively.

3.4.2 ViST (Top-down Sequence Indexes)

The ViST (Virtual Suffix Tree) index structure is proposed by Wang, Park, et al. (2003). Before we illustrate an example of ViST, please note that the data-tree in Figure 15 (B) is an encoded form of the data-tree in Figure 15 (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 15 (A) is the same as the example edge-labeled data-tree in Figure 2. As an example of ViST, consider the data-tree in Figure 15 (B) and the query tree in Figure 15 (D). Both trees are transformed into structure-encoded sequences as illustrated below. Note that each bracket contains two parts. The first part contains the elements' tag. The second part contains the root path of the parent node of the node listed in the first part.

Data Tree 2 (D2) : (F,0) (B,F) (A,FB) (P,F) (P,F) (A,FP) (P,F) (R,FP) (A,FP)
Query (Q) : (F,0) (P,F) (R,FP) (A,FP)

The underlined subsequences of "D2" match the underlined subsequence of "Q". So, we return the matched subsequence in the data-tree as an answer to the query. Nevertheless, we should be aware of any existing false-positives (a.k.a. false alarm) in the solution that may take place. For example, consider the data-tree 3 in Figure 15 (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 "Q" over the "D3" data, we notice that the underlined sequence forms an answer for the query "Q" above, however, it is not a correct answer for the given query because the "R" and the "A" child nodes do not have the same parent "P" node. This is an example of a false-positive answer.

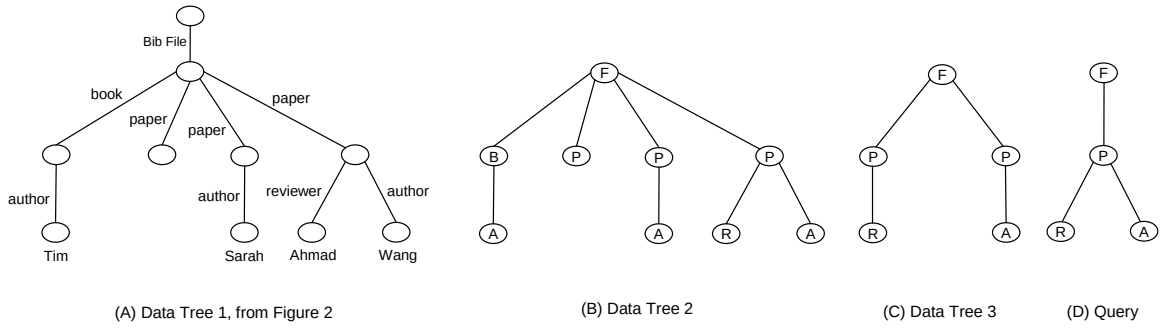


Figure 15. Data trees and a query

In addition to false-positives, the sequence schemes also have the problem of false-negatives (a.k.a. false dismissals), 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 16, 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 solve 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.

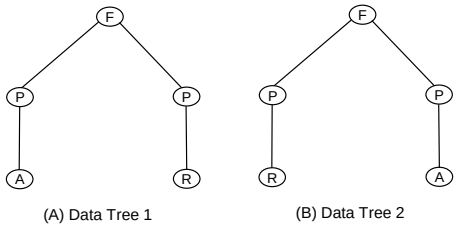


Figure 16. Example of false dismissal

ViST, as we noticed above, 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 is based on B+-tree (Wang, Park, et al., 2003), which is physically implemented as two levels of B+-trees (Gou & Chirkova, 2007). 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.

3.4.3 PRIX (Bottom-up Sequence Indexes)

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 (2004) 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 gets higher at the bottom. That is why it is more efficient to index a tree using the bottom-up traversal direction instead of using the top-down traversal direction. PRIX's bottom-up indexing scheme is a major source of improvement over ViST schemes (Rao & Moon,2004).

PRIX is based on Prufer sequences. To illustrate how a Prufer sequence is used to denote a graph tree, we use the data-tree in Figure 17, which is the same as the data-tree in Figure 15 (B). The letters inside the node circles represent the tag types (names) and the numbers shown beside the nodes represent the post-order numbering of the tree. To encode the tree in Figure 17 with a Prufer sequence, we repeatedly delete the leaf node that has the smallest label and append the label of its parent to the sequence.

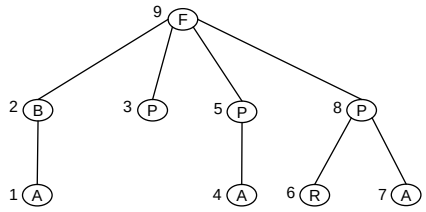


Figure 17. An example of Prufer sequence

As we can see in Figure 17, the smallest label 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}. We then delete label number "3" and add label "9" to the sequence, so the sequence will change to {2,9,9} 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 will be processed more efficiently and faster 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 the false-positives and the false-negatives that are associated with sequence index schemes. Basically, PRIX overcomes the false-positives problem by using document by document post-processing which is time consuming process.

PRIX is based on the B+tree, and it is built in a way similar to ViST (Rao & Moon, 2004). 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.

3.4.4 Summary of Sequence Indexes

Table 4 includes a summary of the sequence indexing schemes. The precision of an index scheme could be 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. Indexing can be implemented in either a top-down direction or bottom-up direction. The types of queries that are supported efficiently by these sequence indexes are both path and twig queries.

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 efficiently**		Path & Twig	Path & Twig
6	**Maintainability**		$O(b\log_b n)$	$O(b\log_b n)$

Table 4. Comparison between Top-down (ViST) and Bottom-up (PRIX) sequencing schemes.

3.5 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 in the one hand, and between the size and the efficiency of the index in answering a query on the other hand. The advantages of one index scheme can be the disadvantages of another, and vice versa. In this section we compare the three categories of structural indexes, namely, node index schemes, graph index schemes, and sequence index schemes.

3.5.1 Criteria for Comparison among Structural Indexing Schemes

In addition to the criteria listed in *Criteria for Comparison among Structural Indexing Schemes* section, the following criteria are used for comparison between the above three types of structural indexing schemes.

- *(A) Computation 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) Computation 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 increases, 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 represented by 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 can not 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 significantly increases the complexity of the XML query evaluation process. In contrast, we can have the values integrated into the structural index. This integration not only saves some additional joins, but it also 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.
- *Main role in answering a query.*
  Each indexing scheme has a different role in the query evaluation process. Some indexes, for example, are used only for joining the elements, while others are used to find a path.

Therefore, different indexing schemes are used for different needs in the query evaluation process.

3.5.2 Comparison among Structural Indexes

Generally, the 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 the sequence indexing scheme to evaluate a query. On the contrary, the node and the graph index always return a complete answer because the order of the nodes is not encoded within the structure of the index as opposed to the sequence 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 path and twig queries. Graph indexes support the path queries without the need for structural joins, but in order to evaluate the twig queries, structural joins are required at the branching node. Finally, the sequence indexes are the best because the structure is encoded within the sequence. Therefore, they do not require any structural joins for path queries 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 paths to solve the query. On the other hand, to evaluate a twig query using a graph index, we break the query into several paths and solve 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, in addition to providing a structural summary, provide valuable utilities for the query optimization. For example, strong DataGuides (Goldman & Widom, 1997) are used in Lore (Abiteboul, Quass, et al., 1997; McHugh, Abiteboul, et al., 1997) to facilitate annotation of sample values and statistical data. The annotated information is associated with the Data Guide objects (nodes). The sample values are used in Lore to provide users with samples of possible values of an element. The number of incoming and outgoing edges for a specific node in a Data

Guide is an example of statistical information that can be annotated. 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 (Cooper et al., 2001; Weigel et al., 2004), although, graph indexes are not designed to carry any values within the structural summary. Node indexes can not 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 5. 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 by these indexing schemes could be path, twig, or both. The maintainability of graph and sequence indexes are 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.

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
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>		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
10- Main role in answering XML query		Path joining	Path selection	Complete query evaluation

Table 5. Summary of comparison among the 3 categories of structural indexing schemes.

3.5.3 Limitations and Open Problems

Despite the extensive research in structural indexes for XML data, there are still many limitations and open problems. XML data can be categorized as semistructured data (Buneman, 1997), which is data that may be irregular or incomplete, and whose structure may change rapidly or unpredictably (McHugh & Widom, 1999). The main challenge in indexing XML data therefore, 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 queries that are used to query XML data 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 numbering index schemes dynamic so that they adapt gracefully to deletion and insertion of new nodes is not an easy task. Node indexes require the highest number of joins among the three indexing schemes to solve an XML query. In order to reduce this shortcoming, it is useful to explore a proper method to optimally use node indexes together with graph indexes to solve XML queries. Each type of index plays a different role. A graph index is used for path selection, whereas a node index is used for path joining (Gou & Chirkova, 2007). In this case, the graph index reflects the structure of data and partitions the data nodes into sets of nodes that share the same structure' characteristics. Then, the node index is used later to reflect the relationship (parent/child and ancestor/descendent) between the individual nodes within these sets. Since the graph index already covers the structure of the indexed data-trees, the node index that can be integrated with the graph index does not have to reflect the structure too. It is sufficient for the integrated node index to maintain a unique identity for each and every node in the indexed data-trees. We can associate these identities with the graph index nodes and then use the graph index along with these identities in the node index to identify the relationship between any two arbitrary nodes. The anticipated node indexing scheme should overcome or minimize the present problems in the existing node schemes.

Node indexes are implemented by two dominant labeling schemes: the prefix (e.g. Dewey) and the interval (e.g., Beg, End) labeling schemes. Each one of these schemes has its own advantages and disadvantages. The size of the interval indexes grows constantly regardless of the data-tree depth, while it grows exponentially in the prefix indexes. Processing time of interval indexes is shorter than that of prefix indexes, because the range labeling scheme is based on numbers while the latter is based on strings. The information of a data-tree paths are included within the prefix labels, while it is not included within the interval labels and require extra processing step to be computed. Prefix indexes are relatively easy to update while interval indexes are harder. A possible research area would be to investigate integrating these two node indexing schemes into one indexing scheme that retains all the desired characteristics in an index. The integrated scheme may have, but should not be limited to, the following characteristics: a small reasonable size; based on numbers (not string); a path can be calculated within a relatively small cost; and easy to be updated.

Backward bisimilar graph indexes can solve simple path queries efficiently. The branching, however, has to be dismantled into multiple sub-queries, where each sub-query is equivalent to a single path in the twig. Expensive join operations are then used to combine these results to create final answers. This problem is solved in forward and backward bisimilar graph indexes as the branching queries are solved as one complete query. 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 solve an XML query with a predicate. Identifying a suitable set of statistics for given graph-based data that can be efficiently computed and stored without having a fixed graph index is an open problem (McHugh & Widom, 1999). A hierarchy of graph covering indexes is yet another open area of research (Kaushik, Bohannon, Naughton, & Korth, 2002). 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 (Gou & Chirkova, 2007). The overhead of the false-positives problem is a major drawback of sequence indexes. Finally, the skipping mechanism needs to be improved as it visits many data nodes needlessly during a query evaluation.

3.5.4 Related Work

Indexing and querying XML data have been active research areas in recent years. Many previous research efforts in the field of information technology have been adapted to index XML data. For example, IR has been used for text-dense XML documents (Xu et al., 2005; Guo et al., 2003). A Suffix Tree has been used by Wang, Park, et al. (2003) to develop dynamic indexes. The Index Fabric by Cooper et al. (2001) is based on the Patricia trie (Knuth, 1998) (a string indexing scheme). The research on optimization of path expressions in object-oriented database systems (Gardarin et al., 1996) and the graph-based semistructured data models (Abiteboul, 1997; Abiteboul, Quass, et al., 1997), have been adapted by McHugh and Widom (1999) in developing Lore (an XML DBMS). Inverted indexes (Salton & McGill, 1983) have been used to support containment queries (Zhang et al., 2001), and to build XML indexes (Dong & Halevy, 2007; Kaushik, Krishnamurthy, Naughton, & Ramakrishnan, 2004). Anatomy of a native XML databases have been discussed by Feinberg (2004).

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 (Buneman, 1997) in general, including XML documents (Abiteboul, Quass, et al., 1997; Buneman, Davidson, Hillebrand, & Suciu,

1996; Fernandez, Florescu, Kang, Levy, & Suciu,1998; Bertino, Rabitti, & Gibbs, 1998). Other systems are designed specifically for XML data (Schoning, 2001; Fiebig et al., 2002; Paparizos et al., 2003; Barta, Consens, & Mendelzon, 2004; Che et al., 2006; Wang, Liu, et al., 2003), or have migrated to a fully XML-based data model (McHugh, Abiteboul, et al., 1997; Goldman, McHugh, et al., 1999; McHugh & Widom, 1999). Finally, there are languages that are designed to query only XML data (Chamberlain, Robie, & Florescu, 2000; Boag et al., 2007; Deutsch, Fernandez, Florescu, Levy, & Suciu, 1998; University of Washington, 2001; Naughton et al., 2001; Robie et al., 1999).

4. CONCLUSIONS

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 the query optimization in the relational databases, XML query optimization is a comparatively new research area. An XML query passes through several stages before it gets completely evaluated. The evaluation process starts with the parsing stage, where the query is converted to a logical query (high level execution strategy query). It is then transformed into several physical plans where most of the optimization is carried out (McHugh & Widom, 1999; De Aguiar, Filho, & Harder, 2006). These plans' costs are estimated by the optimizer and the cheapest plan is executed by the low-level query execution engine.

A key factor in improving the XML queries is indexing (Zou et al., 2004). 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 the path queries as well as the twig queries. A twig query consists of two parts: (1) the structural part, which is specified by the twig branches; (2) the values that are associated with the branches.

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 simple path queries. They evaluate a simple path query by traversing one path only. In contrast, non-deterministic graph indexes may traverse more than one index path to solve a simple 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. We use common criteria to analyze the characteristics of the most common types of 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. 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, which can be used during a query optimization process. Some indexes cover twig and path queries, while others cover only 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).□

ACKNOWLEDGMENT

This work was supported by the Natural Science and Engineering Research Council of Canada (NSERC).

REFERENCES

Abiteboul. S. (1997, January 8-10). Querying semistructured data. In F.N. Afrati, P.G. Kolaitis (Eds.), *Proceedings of the International Conference on Database Theory, ICDT'97,* Delphi, Greece (LNCS 1186, pp.1–18). London, UK: Springer-Verlag.

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

Abiteboul, S., Quass, D., McHugh, J., Widom, J., &.Wiener. J. (1997, April). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1), 68-88.

Al-Khalifa, S., Jagadish, H. V., Koudas, N., Patel, J. M., Srivastava, D., & Wu, Y. (2002, February 26-March 1). Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In R. Agrawal, K. Dittrich, & A.H.H. Ngu (Eds.), *Proceedings of the 18th International Conference on Data Engineering,* San Jose, CA, USA (pp.141-154). Los Alamitos, CA, USA: IEEE Computer Society.

Amagasa, T., Yoshikawa, M., & Uemrua, S. (2003, March 5-8). QRS: A Robust Numbering Scheme for XML Documents. In U. Dayal, K. Ramamritham, & T.M. Vijayaraman (Eds.), *Proceedings of the 19th International Conference on Data Engineering.* Bangalore, India (pp. 705-707). IEEE Computer Society.

Amer-Yahia, S., Baeza-Yates, R., Consens, M., & Lalmas, M. (2007, September 23-27). XML Retrieval: DB/IR in Theory, Web in practice. In C. Koch, J. Gehrke, M. Garofalakls, D. Srlvastava, K. Aberer, A. Deshpande, et al. (Eds.). *Proceeding of the 33rd International Conference on Very Large Data Bases,* Vienna, Austria (pp. 1437-1438). VLDB Endowment.

Baeza-Yates, R., Consens, M. (2004, August 31 - September 3). The Continued Saga of DB-IR Integration. In M.A. Nascimento, M.T. Ozsu, D. Kossmann, R.J. Miller, J.A. Blakeley, & K.B. Schiefer (Eds.). *Proceedings of the 30th VLDB Conference*. Toronto, Canada (pp. 1245-1246). San Fransisco, CA, USA: Morgan Kaufmann.

Barta, A., Consens, M., & Mendelzon, A. (2004, June 11-18). XML Query Optimization Using Path Indexes. In I. Manolescu, & Y. Papakonstantinou (Eds.), *Proceedings of the First International Workshop on XQuery Implementation, Experience, and Perspectives, in cooperation with ACM SIGMOD,* Paris, France (pp.43-48).

Bary, T., Paoli, J., & Sperberg-McQueen, C.M. (Eds.). (1998, February 10). *Extensible Markup language (XML) 1.0*. Retrieved January 22, 2009, from http://www.w3.org/TR/1998/REC-xml-19980210.html.

Bertino, E., Rabitti, F., & Gibbs. S. (1998, January). Query processing in a multimedia document system. A*CM Transactions on Office Information Systems,* 6(1), 1–41.

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

Bruno, N., Koudas, N., & Srivastava, D. (2002, June 3-6). Holistic Twig Joins: Optimal XML Pattern Matching. *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* Madison, Wisconsin, USA (pp.310-321). New York, NY, USA: ACM Press.

Buneman, P. (1997, May 11-15). Semistructured data. *In Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems,* Tucson, Arizona, USA (pp. 117–121). New York, NY, USA: ACM Press.

Buneman, P., Davidson, S., Hillebrand, G., & Suciu. D. (1996, June 4-6). A query language and optimization techniques for unstructured data. In J. Widom (Ed.), *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Montreal, Quebec, Canada (pp.505-516). New York, NY, USA: ACM Press.

Carey, P. (2004). *New Perspective on XML-Comprehensive.* Boston, Massachusetts, USA: Course Technology.

Chamberlain, D., Robie, J., & Florescu, D. (2000, May 18-19). Quilt: An XML query language for heterogeneous data sources. In G. Goos, J. Hartmanis, & J. van Leeuwen (Eds.), *The World Wide Web and Databases, Third International Workshop, WebDB 2000,* Dallas, Texas, USA (LNCS 1997,pp.1-25) Berlin, Germany: Springer.

Che, D., Aberer, K., & Ozsu, M.T. (2006, September). Query optimization in XML structured-document databases. *The VLDB Journal,* 15(3), 263-289.

Chen, Q., Lim, A., & Ong, K. (2003, June 9-12). D(k)-Index: An adaptive Structural summary for graph-structured data. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA (pp.134-144). New York, NY, USA: ACM Press.

Chen, Y., Mihaila, G.A., Bordawekar, R., & Padmanabhan, S. (2004, March 14-18). L-Tree: a Dynamic Labeling Structure for Ordered XML Data. In W. Lindner, M. Mesiti, C. Turker, Y. Tzizikas, & A. Vakali (Eds.). *Current Trends in Database Technology - EDBT 2004 Workshops,* Herakleion, Greece (LNCS 3268, pp. 209-218). Germany, Berlin: Springer.

Chien, S., Vagena, Z., Zhang, D., Tsotras, V., & Zaniolo, C. (2002, August 20-23). Efficient structural joins on indexed XML documents. In P.A. Bernstein, Y.E. Ioannidis, R. Ramakrishnan, & D. Papadias (Eds.), *Proceedings of 28th International Conference on Very Large Data Bases,* Hong Kong, China (pp.263-274). San Fransisco, CA, USA: Morgan Kaufmann.

Chung, C., Min, J., & Shim, K. (2002, June 3-6). APEX: An Adaptive Path Index for XML data. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA (pp.121-132). New York, NY, USA: ACM Press.

Clark, J., & DeRose, S. (Eds.). (1999, November 16). *XML Path Language (XPath) Version 1.0.* Retrieved January 22, 2009, from http://www.w3.org/TR/xpath.

Cohen, E., Kaplan, H., & Milo, T. (2002, June 3-5). Labeling Dynamic XML Trees. *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems,* Madison, Wisconsin, USA (pp. 271-281) New York, NY, USA: ACM Press.

Cooper, B.,Sample, N., Franklin, M., Hjaltason, G., & Shadmon. M. (2001, September 11-14). A Fast Index for Semistructured Data. In P.M.G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, & R.T. Snodgrass (Eds.), *Proceedings of 27th International Conference on Very Large Data Bases VLDB,* Roma, Italy (pp.341-350). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

de Aguiar, J., Filho, M., & Harder, T. (2006, June 6-9). Statistics for Cost-Based XML Query Optimization. In S. Brass, & A. Hinneburg (Eds.), *18th GI-Workshop on the Foundations of Databases (Tagungsband zum 18. GI-Workshop über Grundlagen von Datenbanken)*, Wittenberg, Sachsen-Anhalt (pp. 110-114). Germany: Institute of Computer Science, Martin-Luther-University.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., & Suciu, D. (1998, August 19). *XML-QL: A query language for XML*. Retrieved January 20, 2009, from http://www.w3.org/TR/NOTE-xml-ql.

Dietz.P. (1982, May 5-7). Maintaining order in a linked list. *In Proceedings of the fourteenth annual ACM symposium on Theory of Computing,* San Francisco, California, USA (pp.122 –127). New York, NY, USA: ACM Press.

Dong, X., & Halevy, A. (2007, June 11-14). Indexing Dataspaces. In C.Y. Chan, B.C. Ooi, & A. Zhou (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Beijing, China (pp.43-54). New York, NY, USA: ACM Press.

Duong, M., & Zhang, Y. (2005, January 31 – February 3). LSDX: A New Labeling Scheme for Dynamically Updating XML Data. In H.E. Williams, & G. Dobbie (Eds.), *Database*

Technologies 2005, Proceedings of 16th Australasian Database Conference, Newcastle, Australia (Vol.39, pp. 185-193). Melbourne, Australia: Victoria University.

Feinberg, G. (2004, November 15-19). 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.

Fernandez, M., Florescu, D., Kang, J., Levy, A., & Suciu. D. (1998, June 2-4). Catching the boat with Strudel: Experiences with a website management system. In A. Tiwary, & M. Franklin, *Proceedings ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA (pp.414– 425). New York, NY, USA: ACM Press.

Fiebig, T., Helmer, S., Kanne, C., Moerkotte, G., Neumann, J., Schiele, R., et al. (2002, November). Anatomy of a native XML base management system. *The VLDB Journal*, 11(4), 292-314.

Fisher, D.K., Lam, F., Shui, W.M., & Wong, R.K.. (2006, January 16-19). Dynamic Labeling Schemes for Ordered XML Based on Type Information. In G. Dobbie, & J. Bailey (Eds.), *Proceedings of the 17th Australasian Database Conference,* Hobart, Australia (Vol. 49, pp. 59-68). Darlinghurst, Australia: Australian computer Society, Inc.

Florescu, D., & Kossmann, D. (1999, September). Storing and querying XML data using an RDMBS. *Bulletin of the Technical Committee on Data Engineering (IEEE-CS),* 22(3), 27-34.

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

Gardarin, G., Gruser, J., & Tang, Z. (1996, September 3-6). Cost-based selection of path expression processing algorithms in object-oriented databases. T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda (Eds.), *Proceedings of the Twenty-Second International Conference on Very Large Data Base,* Bombay, India (pp. 390–401). San Fransisco, CA, USA: Morgan Kaufmann.

Goldman, R., McHugh, J., & Widom, J. (1999, June 3-4). From semistructured data to XML: Migrating the Lore data model and query language. In S. Cluet, & T. Milo (Eds.), *Proceedings of the 2nd International Workshop on the Web and Databases, ACM SIGMOD Workshop,* Philadelphia, Pennsylvania, USA (pp. 25-30).

Goldman, R., & Widom, J. (1997, August 25-29). DataGuides: Enabling query formulation and optimization in semistructured databases. In M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, & M.A. Jeusfeld (Eds.), *Proceedings of 23rd International Conference on Very Large Data Bases, VLDB'97,* Athens, Greece (pp.436-445). San Fransisco, CA, USA: Morgan Kaufmann Publishers Inc.

Goldman, R., & Widom, J. (1999, January 13). Approximate Data Guide. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, Israel.

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

Grust, T. (2002, June 3-6). Accelerating XPath location steps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison. Wisconsin, USA (pp. 109-120). New York, NY, USA: ACM Press.

Guo, L., Shao, F., Botev, C., & Shanmugasundaram, J. (2003, June 9-12). XRANK: Ranked keyword search over XML documents. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*. San Diego, California, USA (pp.16-27). New York, NY, USA: ACM Press.

Jiang, H., Lu, H., Wang, W., & Chin Ooi, B. (2003, March 5-8). XR-tree: Indexing XML data for efficient structural joins. In U. Dayal, K. Ramamritham, & T.M. Vijayaraman (Eds.), *Proceedings of the 19th International Conference on Data Engineering.* Bangalore, India (pp.253-263). IEEE Computer Society.

Kaushik, R., Bohannon, P., Naughton, J., & Korth, H. (2002, June 3-6). Covering indexes for branching path queries. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA (pp.133-144). New York, NY, USA: ACM Press.

Kaushik, R., Bohannon, P., Naughton, J., & Shenoy, P. (2002, August 20-23). Updates for Structure Indexes. In P.A. Bernstein, Y.E. Ioannidis, R. Ramakrishnan, & D. Papadias (Eds.), *Proceedings of 28th International Conference on Very Large Data Bases,* Hong Kong, China (pp.239-250). San Fransisco, CA, USA: Morgan Kaufmann.

Kaushik, R., Krishnamurthy, R., Naughton, J., & Ramakrishnan, R. (2004 June 13-18). On the integration of structure indexes and inverted lists. In G. Welkum, A.C. Konig, & S. Dessloch (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France (pp.779-790). New York, NY, USA: ACM Press.

Kaushik, R., Shenoy, P., Bohannon, P., & Gudes, E. (2002, 26 February - 1 March). Exploiting local similarity for indexing paths in graph-structured data. In A.D. Williams, & S. Kawada (Eds.), *Proceedings of 18th International Conference on Data Engineering,* San Jose, California, USA (pp. 129-140). Los Alamitos, CA, USA: IEEE Computer Society.

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

Li, H., Lee, M.L., Hsu, W., & Chen, C. (2004, September). An Evaluation of XML Indexes For Structural Join. *ACM SIGMOD Record*, 33(3), 28-33.

Li, Q., & Moon, B. (2001, September 11-14). Indexing and querying XML data for regular path expressions. In P.M.G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, & R.T. Snodgrass (Eds.), *Proceedings of 27th International Conference on Very Large Data Bases,* Roma, Italy (pp.361-370). San Fransisco, CA, USA: Morgan Kaufmann.

Lu, J., Ling, T., Chan, C., & Chen, T. (2005, August 30-September 2). From region encoding to extended Dewey: On efficient processing of XML twig pattern matching. In K. Bohm, C.S. Jensen, L.M. Haas, M.L. Kersten, P. Larson, & B.C. Chin (Eds.), *Proceedings of the 31st International Conference on Very Large Data Bases. VLDB*, Trondheim, Norway (pp. 193-204). New York: ACM Press.

Lu, J., & Ling, W. (2004, April 14-17). Labeling and Querying Dynamic XML Trees. In J. Xu Yu, X. Lin, H. Lu, & Y. Zhang (Eds.), *Advanced Web Technologies and Applications, 6th Asia-Pacific Web Conference,* Hangzhou, China (LNCS 3007, pp. 180-189). Berlin, Germany: Springer.

Mariano, P., & Baeza-Yates, R. (2005, December 7-9). Database and Information retrieval Techniques for XML. In S. Grumbach, L. Sui, & V. Vianu (Eds.). *Advances in Computer Science-ASIAN 2005, Data Management on the Web, 10th Asian Computing Science Conference*, Kunming, China (LNCS 1318, pp. 22-27). Berlin, Germany: Springer.

McHugh, J., Abiteboul, S., Goldman, R., Quass, D., & Widom, J. (1997, September). Lore: A database management system for semistructured data. *ACM SIGMOD Record,* 26(3), 54–66.

McHugh, J., & Widom, J. (1999, September 7-10). Query Optimization for XML. In M.P. Atkinson, M.E. Orlowska, P. Valduriez, S.B. Zdonik, & M.L. Brodie (Eds.), *Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99,* Edinburgh, Scotland, UK (pp.315-326). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Megginson, D., & Brownell, D. (2004, April 27). *Simple API for XML (SAX).* Retrieved January 22, 2009, from http://www.saxproject.org/.

Milo, T., & Suciu, D. (1999, January 10-12). Index Structures for Path Expressions. In C. Beeri, & P. Buneman (Eds.), *Database Theory –ICDT'99, Proceedings of 7th International Conference on Database Theory*, Jerusalem, Israel (LNCS 1540, pp.277-295). Berlin, Germany: Springer.

Moro, M., Vagena, Z., & Tsotras, V. (2005, August 30 - September 2). Tree-Pattern Queries on a Lightweight XML Processor. In K. Bohm, C.S. Jensen, L.M. Haas, M.L. Kersten, P. Larson, & B.C. Chin (Eds.), *Proceedings of the 31st International Conference on Very Large Data Bases. VLDB*, Trondheim, Norway (pp. 205-216). New York: ACM Press.

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

O'Neil, P., O'Neil, E., Pal, S., Cseri, I., Schaller, G., & Westbury, N. (2004, June 13-18). ORDPATHs: Insert-Friendly XML Node Labels. In G. Welkum, A.C. Konig, & S. Dessloch (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Paris, France (pp. 903-908). New York, NY, USA: ACM Press.

Online Computer Library Center. (2008). *Dewey decimal classification*. Retrieved January 13, 2009, from http://www.oclc.org/dewey/versions/ddc22print/intro.pdf.

Paparizos, S., Jagadis, H., Patel, J., Al-Khalifa, S., Ladshmanan, L., Srivastava, D., et al. (2003, June 9-12). TIMBER: A native system for querying XML. *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Diego, California, USA. (pp. 672-672). New York, NY, USA: ACM Press.

Polyzotis, N., & Garofalakis, M. (2002, August 20-23). Structure and Value Synopses for XML Data Graphs. In P.A. Bernstein, Y.E. Ioannidis, R. Ramakrishnan, & D. Papadias (Eds.), *Proceedings of 28th International Conference on Very Large Data Bases,* Hong Kong, China (pp. 466-477). San Fransisco, CA, USA: Morgan Kaufmann.

Polyzotis, N., Garofalakis, M., & Ioannidis, Y. (2004, June 13-18). Approximate XML Query Answers. In G. Welkum, A.C. Konig, & S. Dessloch (Eds.), *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Paris, France (pp.263-274). New York, NY, USA: ACM Press.

Rao, P., & Moon. B. (2004, March 30-April 2). PRIX: Indexing and querying XML using Prufer sequences. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004,* Boston, MA, USA (pp.288-300). IEEE Computer Society.

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

Salton, G., & McGill, M.J. (1983). *Introduction to Modern Information Retrieval* (pp. 16-21). New York, NY, USA: McGraw-Hill.

Schoning, H. (2001, April 2-6). Tamino – a DBMS Designed for XML. In D.C. Young (Eds.), Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany (pp. 149-154). Los Alamitos, CA, USA: IEEE Computer Society.

Silberstein, A., He, H., Yi, K., & Yang, J. (2005, April 5-8). BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proceeding of the 21st International Conference on Data Engineering, ICDE 2005,* Tokyo, Japan (pp.285-296). Washington, DC, USA: IEEE Computer Society.

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

Tatarinov, I., Viglas, S., Beyer, K., Shanmugasundaram, J., Shekita, E., & Zhang. C. (2002, June 3-6). Storing and Querying Ordered XML Using a relational Database System. *In Proceedings of the ACM SIGMOD International Conference on Management of Data,*. Madison, Wisconsin, USA (pp.204-215). New York, NY, USA: ACM Press.

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

Thompson, H.S., Beech, D., Maloney, M., & Mendelsohn, N. (Eds.). (2004, October 28). *XML Schema Part 1: Structures Second Edition.* Retrieved January 26, 2009, from http://www.w3.org/TR/xmlschema-1/.

University of Washington. (2001, May 10). *The Tukwila system.* Retrieved January 14, 2009, from http://xml.coverpages.org/tukwila.html.

Vakali, A., Catania, B., & Maddalena, A. (2005, March-April). XML Data Stores: Emerging Practices. *Internet Computing, IEEE*. 9(2),62-69.

Wang, G., Liu, M., Sun, B., Yu, G., Lv, J., Xu Yu, J., et al. (2003, July 16-18). Effective Schema-Based XML Query Optimization Techniques. In B.C. Desai, & W. Ng (Eds.), *Proceedings of the 7th International Database Engineering and Applications Symposium. IDEAS'03,* Hong Kong, China (pp. 230-235). Los Alamitos, CA, USA: IEEE Computer Society.

Wang, H., & Meng, X. (2005, April 5-8). On the sequencing of tree structures for XML indexing. In *Proceeding of the 21st International Conference on Data Engineering, ICDE 2005,* Tokyo, Japan (pp. 372-383). Washington, DC, USA: IEEE Computer Society.

Wang, H., Park, S., Fan, W., & Yu., P. (2003, June 9-12). ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. *In Proceedings of the ACM SIGMOD International Conference on Management of Data,* San Diego, California, USA (pp.110-121) . New York, NY, USA: ACM Press.

Wang, W., Wang, H., Lu, H., Jiang, H., Lin, X., & Li, J. (2005, August 30-September 2). Efficient Processing of XML Path Queries Using the Disk-based F&B Index. In K. Bohm, C.S. Jensen, L.M. Haas, M.L. Kersten, P. Larson, & B.C. Chin (Eds.), *Proceedings of the 31st International Conference on Very Large Data Bases. VLDB*, Trondheim, Norway (pp. 145-156). New York: ACM Press.

Weigel, F., Meuss, H., Bry, F., & Schulz, K.U. (2004, April 5-7). Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In S. McDonald, & J. Tait (Eds.), *Advances in Information Retrieval, Proceedings of 26th European Conference on Information Retrieval, ECIR 2004*. Sunderlank, UK. (LNCS 2997, pp. 378-393). Berlin, Germany: Springer.

Wu, X., Lee, M., & Hsu, W. (2004, March 30-April 2). A Prime Number Labeling Schemes for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004,* Boston, MA, USA (pp.66-78). IEEE Computer Society.

Xu, Y., & Papakonstantinou, Y. (2005, June 14-16). Efficient keyword search for smallest LCAs in XML databases. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA (pp. 527–538). New York, NY, USA: ACM press.

Yang, B., Fontoura, M., Shekita, M., Rajagopalan, S., & Beyer, K. (2004, November 8-13). Virtual Cursors for XML Joins. In D.A. Evans, L. Gravano, O. Herzog, C. Zhai, & M. Ronthaler, (Eds.), *Proceedings of the thirteenth ACM International Conference on Information and Knowledge Management,CIKM 2004,* Washington, DC, USA (pp.523-532). New York, NY, USA: ACM Press.

Zhang, C., Naughton, R., Dewitt, D., Luo, Q., & Lohman, G. (2001, May 21-24). On Supporting containment Queries in Relational Database Management Systems. In T. Sellis (Ed.), *Proceedings of ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, USA (pp.425-436). New York, NY, USA: ACM Press.

Zou, Q., Liu, S., & Chu, W. (2004, November 12-13). Ctree : A Compact Tree for Indexing XML Data. In A.H.F. Laender, D. Lee, & M. Ronthaler (Eds.), *Proceedings of the 6th annual ACM international workshop on Web Information and Data Management,WIDM 2004,* Washington, DC, USA (pp.39-46). New York, NY, USA: ACM Press.

ADDITIONAL READING

Ali, M.S., Consens, M., Gu, X., Kanza, Y., rizzolo, F., & Stasiu, R. (2007, August). Efficient, Effective and Flexible XML Retrieval Using Summaries. In N. Fuhr, M. Lalmas, & A. Trotman (Eds.), *Comparative Evaluation of XML Information Retrieval Systems* (LNCS 4518, pp. 89-103). Berlin: Germany: Springer.

Alstrup, S., Bille, P., & Rauhe, T. (2003, January 12-13). Labeling Schemes for Small Distances in Trees. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Mathematics*, Baltimore, Maryland, USA (pp. 689-698). Society for Industrial and Applied Mathematics.

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

Barta, A., Consens, m. Mendelzon, A. (2005, August 30-September 2). Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access methods. In K. Bohm, C. Jensen, L. Hass, M. Kersten, P. Larson, & B. Ooi (Eds.), *Proceedings of 31st International Conference on Very Large Data Bases*, Tondheim, Norway (pp.133-144). VLDB Endowment.

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

Catania, B., Maddalena, A., & Vakali, A. (2005, September-October). XML Document Indexes: A Classification. IEEE Internet Computing, 9(5), 64-71.

Catania, B., Ooi, B., Wang, W., & Wang, X. (2005, June 14-16). Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA (pp. 515-526). New York, NY, USA: ACM Press.

Christophides, V., Plexousakis, D., Scholl, M., & Tourtounis, S. (2003 May 20-24). On Labeling Schemes for the Semantic Web. In *Proceedings of the 12th International conference on World Wide Web*, Budapest, Hungary (pp. 544-555). New York, NY, USA: ACM Press

Consens, M., Rizzolo, F., & Vaisman, A. (2008, April 7-12). AxPRE Summaries: Exploring the (Semi-)Structure of XML Web Collections. In *Proceedings of the 24th International Conference on Data Engineering (ICDE'08)*, Cancun, Mexico (pp. 1519-1521). IEEE.

Elghandour, I., Aboulnaga, A., Zilio, D.C., Chiang, F., Balmin, A., Beyer, K., & Zuzarte, C. (2008, June 9-12). An XML Index Advisor for DB2. In J.T. Wang (Ed.), *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, Vancouver, BC, Canada (pp. 1267-1270). New York, NY, USA: ACM Press.

Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A.N., Tian, F., et al. (2003, September 9-12). Mixed Mode XML Query Processing. In J.C. Freytag, P.C. Lockemann, S. Abiteboul, M.J. Carey, P.G. Selinger, & A. Heuer (Eds.), *Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany (pp. 225-236). San Fransisco, CA, USA: Morgan Kaufmann Publishers Inc.

Harder, T., Haustein, M., Mathis, C., & Wagner, M. (2007, January). Node labeling schemes for dynamic XML documents reconsidered. *Data & Knowledge Engineering*, 60(1), 126-149.

Harding, P.J., Li, Q., & Moon, B. (2003, September 9-12). XISS/R: XML Indexing and Storage system Using RDBMS. In J.C. Freytag, P.C. Lockemann, S. Abiteboul, M.J. Carey, P.G. Selinger, & A. Heuer (Eds.), *Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany (pp. 1073-1076). San Fransisco, CA, USA: Morgan Kaufmann Publishers Inc.

Haw, S., Lee, C. (2008, February 17-20). Evolution of Structural Path Indexing Techniques in XML Databases: A Survey and Open Discussion. In *Proceedings of International Conference on Advanced Communication Technology (ICACT'08),* Phoenix Park, Korea (Vol. 3, pp. 2054-2059). Piscataway, NJ, USA: IEEE Computer Society.

Kaplan, H., Milo, T., & Shabo, R. (2002, January 6-8). A comparison of labeling schemes for ancestor queries. In *Proceedings of the 13th annual ACM-SIAM Symposium on Discrete Algorithms*, San Fransisco, CA, USA (pp. 954-963). Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Kha, D., Yoshikawa, M., & Uemura, S. (2001, April 2-6). An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany (pp. 313-320). Los Alamitos, CA, USA: IEEE Computer Society.

Kobayashi, M. & Takeda, K. (2000, June). Information Retrieval on the Web. A*CM Computer Surveys*, 32(2), 144-173.

Krishnamurthy, R., Kaushik, R., & Naughton, J.F. (2003,). XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In Z. Bellahsene, A.B. Chaudhri, E. Rahm, M. Rys, & R. Unland (Eds.), *Database and XML Technologies, First International XML Database Symposium, XSym 2003*, Berlin, Germany (LNCS. 2824, pp. 1-18). Berlin, Germany: Springer.

Li, C., & Ling T.W. (2005, October 31-November 5). QED: A Novel Quaternary Encoding to completely Avoid Re-labeling in XML Updates. In O. Herzog, H. Schek, N. Fuhr, A. Chowdhury, & W. Teiken (Eds.), *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM)*, Bermen, Germany (pp. 501-508). New York, NY, USA: ACM Press.

Li, Y., Yi, P., & Li, Q. (2005, October 12-18). Optimizing Path Expression Queries of XML Data. In *Proceedings of 2005 IEEE International conference on e-Business Engineering (ICEBE'05)*, Beijing, China (pp. 497-504). Los Alamitos, CA, USA: IEEE Computer Society.

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

Lv, J., Wang, G., Yu, J., & Yu, G. (2002, August 11-13). Performance Evaluation of a DOM-Based XML Database: Storage, Indexing, and Query Optimization. In X. Meng, J. Su, & Y. Wang (Eds.), *Advances in Web-Age Information Management, Third International Conference, WAIM 2002,* Beijing, China (LNCS 2419, pp. 13-24). Berlin, Germany: Springer.

Mendelzon, A., Rizzolo, F., & Vaisman, A. (2004, August 31-September 3). Indexing Temporal XML Documents. In M.A. Nascimento, M.T. Ozsu, D. Kossmann, R.J. Miller, J.A. Blakeley, & K.B. Schiefer (Eds.). *Proceedings of the 30th VLDB Conference*. Toronto, Canada (pp. 216-227). San Fransisco, CA, USA: Morgan Kaufmann.

Miler, J., & Sheth, S. (2000, February-March). Querying XML Documents. *IEEE Potentials Magazine,* 19(1), 24-26.

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.

Shalem, M., & Bar-Yossef, Z. (2008, April 7-12). The Space complexity of Processing XML Twig Queries Over Indexed Documents. In *Proceedings of the 24th International Conference on Data Engineering (ICDE'08)*, Cancun, Mexico (pp. 824-832). IEEE.

Vagena, Z., Moro, M., & Tsotras, V. (2004, June 17-18). Twig Query Processing over Graph-Structured XML Data. In S. Amer-Yahia, & L. Gravano (Eds.), *Proceedings of the Seventh International workshop on the Web and Databases (WebDB 2004),* Paris, France (pp. 43-48). New York, NY, USA: ACM Press.

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

Weigel, F., Schulz, K.U., & Meuss, H. (2005, November 5). Exploiting Native XML Indexing Techniques for XML Retrieval in relational Database Systems. In A. Bonifi, D. Lee, & M. Rotnthaler (Eds.), *Proceedings of the Seventh ACM International Workshop on Web Information and Data Management*, Bremen, Germany (pp. 23-30). New York, NY, USA: ACM Press

Yi, K., He, H., Stanoi, I., & Yang, J. (2004, June 13-18). Incremental maintenance of XML Structural Indexes. In G. Weikum, A.C. Konig, & S. DeBloch (Eds.), *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Paris, France (pp. 491-502). New York, NY, USA: ACM Press.

Zhang, N. (2004, March 14-18). XML Query Processing and Optimization. In W. Lindner, M. Mesiti, C. Turker, Y. Tzizikas, & A. Vakali (Eds.). *Current Trends in Database Technology - EDBT 2004 Workshops,* Herakleion, Greece (LNCS 3268, pp. 121-132). Germany, Berlin: Springer.

Zhang, B., Wang, W., Wang, X., & Zhou, A. (2007, April 9-12). AB-Index: An Efficient Adaptive Index for Branching XML Queries. In *Advances in Databases: Concepts, Systems and Applications, Proceedings on the 12th International Conference on Database Systems for Advanced applications, DASFAA 2007,* Bangkok, Thailand,.(LNCS 4443, pp. 988-993). Berlin, Germany: Springer.

Zhang, N., Kacholia, V., & Ozsu, M.T. (2004, March 30-April 2). A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004,* Boston, MA, USA (pp.54-65). IEEE Computer Society.

Zuopeng, L., Kongfa, H., Ning, Y., & Yisheng, D. (2005, October 6). An Efficient Index structure for XML Based on Generalized Suffix tree. *Information Systems*, 32(2), 283-294.