Efficiently Querying Large XML Data Repositories: A Survey

Gang Gou and Rada Chirkova

Abstract—Extensible Markup Language (XML) is emerging as a de facto standard for information exchange among various applications on the World Wide Web. There has been a growing need for developing high-performance techniques to query large XML data repositories efficiently. One important problem in XML query processing is *twig pattern matching*, that is, finding in an XML data tree D all matches that satisfy a specified twig (or path) query pattern Q. In this survey, we review, classify, and compare major techniques for twig pattern matching.[1] Specifically, we consider two classes of major XML query processing techniques: the *relational approach* and the *native approach*. The relational approach directly utilizes existing relational database systems to store and query XML data, which enables the use of all important techniques that have been developed for relational databases, whereas in the native approach, specialized storage and query processing systems tailored for XML data are developed from scratch to further improve XML query performance. As implied by existing work, XML data querying and management are developing in the direction of integrating the relational approach with the native approach, which could result in higher query processing performance and also significantly reduce system reengineering costs.

Index Terms—XML query processing, twig pattern matching.

1 Introduction

Extensible Markup Language (XML) is emerging as a de facto standard for information exchange among various applications on the World Wide Web due to XML's inherent data self-describing capability and flexibility of organizing data [25]. There has been a growing need for developing high-performance techniques to query large XML data repositories efficiently.

First, data in XML documents are self-describing. Similar to the popular Hypertext Markup Language (HTML), XML is based on so-called nested tags. Fig. 1a shows an example of an XML document, which records information about publishers. However, unlike HTML, in which tags associated with data express the presentation style (for example, font style) of data, tags in XML describe the semantics of data. For example, Lines 1–3 in Fig. 1a say that "$Cambridge$" is an address of a publisher whose name is "$MIT\ Press$." This self-describing capability of XML data helps applications on the Web "understand" the content of XML documents published by other applications.

1. The most recent literature discussed in this survey was published in the *Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE '06)*. Given the vast amount of literature on XML, it is difficult to cover everything. We chose to focus on reviewing major techniques for twig pattern matching, which is an issue of major importance in XML query processing. Due to the space limit, we had to omit some other important topics related to XML query processing such as XML publishing (of relational data), answering XML queries by using XML views, and minimizing XML queries.

• *The authors are with the Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206.*
E-mail: ggou@ncsu.edu, chirkova@csc.ncsu.edu.

Manuscript received 3 Mar. 2006; revised 9 Nov. 2006; accepted 13 Apr. 2007; published online 2 May 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0109-0306.
Digital Object Identifier no. 10.1109/TKDE.2007.1060.

Second, XML is flexible in organizing data. The hierarchy formed by nested tags structures the content of XML documents. The role of nested tags in XML is somewhat similar to that of schemas in relational databases. At the same time, the nested XML model is far more flexible than the flat relational model. In an XML document, objects of the same type might have different types of subobjects or different numbers of subobjects of the same type. For example, in Fig. 1a, the first $publisher$, but not the second $publisher$, has an $address$ subelement. The $book$ under the first $publisher$ has two $author$ subelements, but the $book$ under the second $publisher$ has only one.

1.1 Data Model

1.1.1 Basic Model: Trees

The basic data model of XML is a labeled and ordered tree. Figs. 2a and 2b show the data tree of the XML document in Fig. 1a (the pair of numbers adorning each node will be discussed in Section 2.1). Fig. 2a is based on the *node-labeled* model, with labels on nodes, and Fig. 2b is based on the *edge-labeled* model, with labels on edges. These two models are equivalent. We discuss XML data trees based on the node-labeled model, and analogous points hold for the edge-labeled model. There are basically three types of nodes in a data tree:

1. *Element nodes (internal nodes).* These correspond to tags in XML documents, for example, $publisher$.
2. *Attribute nodes (internal nodes).* These correspond to attributes associated with tags in XML documents, for example, "$@name$." In contrast to element nodes, attribute nodes are not nested (that is, an attribute cannot have any subelements), are not repeatable (that is, two same-name attributes cannot occur under one element), and are unordered (that is,

```
<Publishers>
  <publisher @name=' MIT Press' >
    <address> Cambridge </address>
    <book>
      <title> Databases </title>
      <author> Tom </author>
      <author> John </author>
    </book>
  </publisher>
  <publisher>
    <book>
      <title> Life </title>
      <author>
        <name> Smith </name>
        <age> 18 </age>
      </author>
    </book>
    <name> NY Press </name>
  </publisher>
</Publishers>                    (a)
```

```
<Publishers>
  <publisher @name=' MIT Press' >
    <address> Cambridge </address>
    <book>
      <title> Databases </title>
      <author friend=1> Tom </author>
      <author id=1 loves=2> John </author>
    </book>
  </publisher>
  <publisher>
    <book>
      <title> Life </title>
      <author id=2 friend=1>
        <name> Smith </author>
        <age> 18 </age>
      </author>
    </book>
    <name> NY Press </name>
  </publisher>
</Publishers>                    (b)
```

Fig. 1. Examples of XML documents. (a) XML document without ID/IDREF. (b) XML document with ID/IDREF.

attributes of an element can freely interchange their occurrence locations under the element).

3. *Value nodes (leaf nodes).* These correspond to data values in XML documents, for example, "*MIT Press*."

Edges in a data tree represent structural relationships between elements, attributes, and values.

Note that in an XML data tree, some nodes with the same name might be nested on the same path. We call this phenomenon *recursion*. For instance, in a *book* data tree [28], multiple *section* nodes might be nested on the same path. Recursion occurs fairly frequently in XML data in practice. Choi [21] investigated 60 document type descriptors (DTDs),[2] among which 35 are recursive. As we shall see in Section 4.1, recursion in XML data significantly increases the complexity of efficiently querying XML data.

1.1.2 The Extended Model: Directed Acyclic Graphs (DAGs) and General Graphs

XML documents allow users to define ID/IDREF attributes of elements, where an *ID attribute* uniquely identifies an element, and *IDREF attributes* refer to other elements that are explicitly identified by their ID attributes. Fig. 1b shows an XML document with ID/IDREF attributes. ID/IDREF attributes increase the flexibility of the XML data model and extend the basic tree model to DAGs or to even more general directed graphs with cycles. Fig. 2c shows an XML data graph with cycles, which corresponds to the XML document in Fig. 1b.

1.2 XML Queries

Unlike (flat) text documents, XML documents have nested structure. Thus, XML queries concern not only the content but also the structure of XML data. Basically, the queries can be formed using *twig patterns*, in which nodes represent the terms that the user is interested in, that is, the content part of queries, and edges represent the structural relationships that the user wants to hold between the terms, that is, the structural part of queries.

We categorize XML queries into two classes: *database-style queries* (Section 1.2.1) and *Information Retrieval (IR)-style queries* (Section 1.2.2). Database-style queries return all query results that precisely match (the content and structure requirements specified by) the queries, which is similar to SQL query semantics in relational databases. On the other hand, IR-style queries allow "imprecise" or "fuzzy" query results, which are ranked based on their relevance to the queries. Only the top-ranked results are returned to users, which is similar to the semantics of *keyword search* queries in the traditional IR [99] context.

2. DTD is a kind of schema of XML data, which shall be discussed further in Section 3.4.

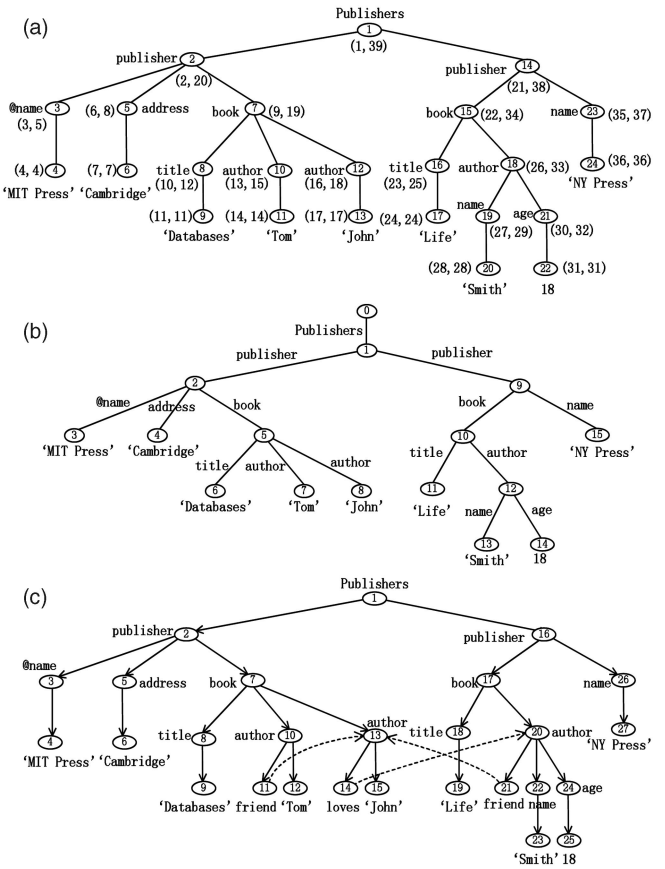


Fig. 2. The XML data model. (a) Node-labeled XML data tree. (b) Edge-labeled XML data tree. (c) Node-labeled XML data graph (with ID/IDREF edges).

1.2.1 Database-Style XML Queries

XML Path Language (XPath) [27] and XQuery [30], originally developed and recommended by the W3C Consortium [25], are today's mainstream (database-style) XML query languages. As we shall see, twig patterns play a very important role in XPath and XQuery.

XPath. XPath [27] is a basic XML query language that selects nodes from XML documents such that the path from the root to each selected node satisfies a specified pattern. A simple XPath query is formulated as a sequence of alternating axes and tags. Two most commonly used axes are the *child axis* "/," where "A/B" denotes selecting B-tagged child nodes of A-tagged nodes, and the *descendant axis* "//," where "$A//B$" denotes selecting B-tagged descendant nodes of A-tagged nodes. Consider an example: An XPath query "$/publisher//title$" would return all $title$ elements under all top-level $publisher$ elements. The result of this query on the data tree in Fig. 2a is two $title$ nodes that have values "Databases" and "Life," respectively.

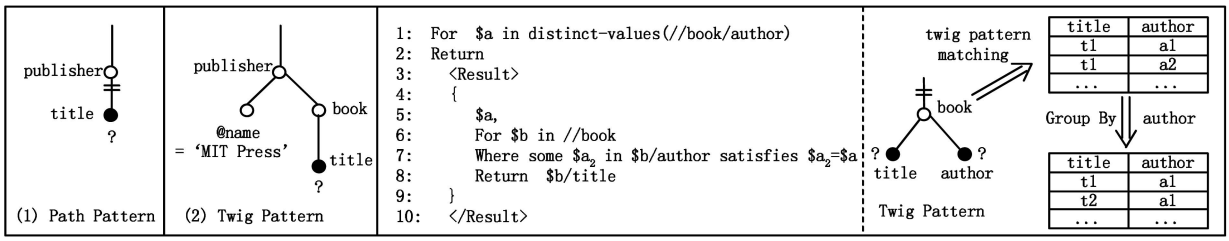


Fig. 3. Examples: (a) XPath and (b) XQuery.

The above XPath query can be formalized as a simple *path pattern*, as shown in Fig. 3a(1), where the single-line edge represents the "/" axis, the edge with "=" represents the "//" axis, and the shaded node is an *output node*. Generally, an XPath query can specify a more complex *twig pattern* by using *predicates* in its expression. One example is "$/publisher[@name = MITPress]/book/title$" (Fig. 3a(2)), in which "$/publisher/book/title$" is the *main path* of the query, and the content between "[" and "]" is a predicate. This query returns all book titles of the publisher with name "$MIT\ Press$." Generally, an XPath query might involve multiple predicates.

From the above, we can see that the core of an XPath query is a twig pattern that includes exactly one output node. Further, we call nodes in query twig patterns *query nodes* and nodes in data trees *data nodes*.

Note that in addition to the *child* axis and the *descendant* axis, XPath also defines 11 other axis types [27]: *parent*, *ancestor*, *descendant-or-self*, *ancestor-or-self*, *following*, *preceding*, *following-sibling*, *preceding-sibling*, *self*, *attribute*, and *namespace*. In this survey, we focus on the "/" and "//" axes, since they are most commonly used in practical applications and have been of the most interest to researchers. Ways to deal with a full set of XPath axes are described in the work of Gottlob et al. [45], [46], [47], [48] and of Grust et al. [50], [52], [53].

XQuery. Query language XQuery [30] is more expressive than XPath. An XQuery query is composed of For-Let-Where-Return (FLWR) clauses, which can be nested and composed with full generality [11]; that is, each clause in itself can include sub–XQuery queries. The *For* and *Let* clauses bind nodes selected by XPath expressions to user-defined *node variables*. The *Where* clauses specify selection or join predicates on node variables. The *Return* clauses operate on node variables to format query results in the XML format (rather than in the simple tuple format, as in the context of answering relational queries). Fig. 3b shows a simple XQuery query, which groups book $titles$ by $author$, on the XML data in Fig. 1a, which instead groups $authors$ by $book$.

Although the nested/compositional syntax of XQuery makes it much more expressive than XPath, the rich semantics also significantly increases the optimization and evaluation complexity of XQuery. Although an XQuery query can be evaluated simply clause by clause by using the nested-loop procedure implied by its syntax, such a naive syntax-driven evaluation plan usually results in poor query performance. For example, whereas the query in Fig. 3b involves four path expressions (in lines 1, 6, 7, and 8, respectively), it is really unnecessary to evaluate these four path expressions over the XML data separately. Indeed, these path expressions have overlapping parts, and evaluating them separately would result in unnecessarily repeated accesses to the same data nodes.

In fact, by carefully analyzing the semantics of the XQuery query in this example, an intelligent XQuery optimizer would be able to find an efficient evaluation plan such as the plan shown on the right side of Fig. 3b. By assembling the four path expressions into a single twig pattern, the plan avoids repeated evaluation of these path expressions. Considerable effort has been devoted to developing such intelligent XQuery optimizers for finding efficient XQuery evaluation plans. For example, the tree algebra for XML (TAX) [61], generalized tree pattern (GTP) [18], and tree logical class (TLC) [91] addressed rewriting XQuery by using a set of predefined tree algebras. Nested XML Tableaux (NEXT) [33] rewrote XQuery by using the so-called NEXT syntax. The NEXT syntax is an XQuery-like syntax, which additionally detects and explicitly expresses the important group-by operator, which is not included in the formal syntax of XQuery but usually exists implicitly in the semantics of many practical XQuery queries (for example, see the query in Fig. 3b). DeHaan et al. [32] and Grust et al. [51] proposed prestoring tree encodings of XML data (see Section 2.1) into relational databases and then rewriting XQuery by using SQL.

Evaluating twig patterns is essential in physical evaluation plans generated by XQuery optimizers. The reason is that XQuery queries use XPath expressions, which are essentially path or twig patterns, in their FLWR clauses to bind or operate on node variables. However, unlike XPath, XQuery does more than evaluating one single simple twig pattern. In particular, 1) answering an XQuery query might involve evaluating multiple twig patterns (although the toy-example query in Fig. 3b evaluates only one), whereas an XPath query evaluates just one. For performance purposes, the number of such twig patterns involved in evaluating XQuery queries should be minimized by XQuery optimizers to avoid repeated evaluation of the same subpath expressions, as discussed earlier. 2) The twig patterns involved in XQuery queries usually include more than one output node (for example, see $title$ and $author$ in the twig pattern in Fig. 3b), whereas the twig pattern of an XPath query has exactly one. 3) Evaluating twig patterns is not the only operation needed for evaluating XQuery queries. To obtain correct query results, other physical operations such

as the *group-by* operation shown on the right side of Fig. 3b usually have to be performed on the results returned by twig pattern matching. Finally, the query results need to be formatted in XML, as specified in the *Return* clauses.

1.2.2 IR-Style XML Queries

IR-style XML queries are mainly used to query *text-dense* XML data repositories whose value elements typically involve long texts. For example, the XML document in Fig. 1a is not text dense, since most of its value elements, for example, "$Databases$," "Tom," and "$John$," include only very short texts. However, this document would become text dense if some new $review$ elements containing long texts were added under $book$ as subelements.

Unlike database-style XML queries, there is no commonly agreed standard language for expressing IR-style XML queries. We now briefly introduce two main classes of IR-style XML queries: DB+IR queries and IR-only queries. Note that unlike traditional IR-style queries, which are performed at the granularity of documents, that is, concerning all documents relevant to queries, these IR-style XML queries are performed at the granularity of XML elements, that is, concerning all XML elements (in XML documents) relevant to queries.

DB+IR queries. DB+IR queries enhance database-style XML queries such as XPath and XQuery queries with IR-style characteristics. For instance, it is common to enhance XPath or XQuery queries with a *contains* function to perform IR-style keyword search. A simple example is "$//publisher[contains(\,"Databases","Tom"\,)]/@name$." It returns the names of all publishers whose (child or descendant) subelements contain approximate matches to keywords "$Databases$" and "Tom." Such $contains$-enhanced DB+IR queries have been considered, for instance, in XIRQL [43], CtreeIR [77], and FleXPath [5]. Li et al. [76] proposed to enhance XQuery with an $mlcas$ function to perform lowest common ancestor (LCA) search (to be discussed shortly).

IR-only queries. The format of IR-only queries is less rigid than that of DB+IR queries: IR-only queries do not have the structural part and specify the content part as a set of keywords $K = \{K_1, K_2, \ldots, K_n\}$ (this is similar to keyword search queries in the traditional IR context). The candidate answers of IR-only queries are from the set of LCAs of those data nodes that correspond to the keywords in K, since LCAs typically represent the most *specific* answer elements that are relevant to the given keywords. For example, an IR-only query $\{"Databases","Tom"\}$ over the data tree in Fig. 2a will retrieve the $book$ node under the first $publisher$ node, whereas the first $publisher$ node is not retrieved although it is a common ancestor of the "$Databases$" and "Tom" nodes. Intuitively, the $book$ element is more specifically relevant to "$Databases$" and "Tom" than the first $publisher$ element. IR-only queries have been considered in, for instance, XRANK [54] and XKSearch [121].

Ranking. In addition to finding candidate query results, IR-style XML queries (both DB+IR queries and IR-only queries) need to rank the candidate query results based on their relevance to the queries and to return to the user only the top-ranked results, as in the traditional IR context. Many relevance measures developed in the traditional IR context such as keyword proximity [54] and weighted term frequency [77] are applicable to building ranking functions in the XML IR context. A very useful relevance measure in the XML IR context is *result specificity* [54], which ranks more specific results higher than less specific results. The motivation is similar to that of computing LCAs for IR-only queries.

Due to the space restriction, in the remaining survey, we address database-style XML queries only. At the same time, many query techniques developed for database-style XML queries are also applicable to IR-style XML queries. For example, Dewey coding (introduced in Section 2.1) plays a key role in computing LCAs for IR-only queries [54], [121].

1.3 Problem Statement

As discussed in Section 1.2.1, twig pattern matching is essential in evaluating XPath/XQuery queries. In fact, it is one of the most important problems in XML query processing. Formally, the problem of twig pattern matching is to find in an XML data tree D all matches that satisfy a given twig (or path) pattern Q. The twig pattern of an XPath query has just one output node, and the result of twig pattern matching is returned as a set of (answer) nodes. The twig patterns involved in an XQuery query usually include more than one output node, and the result of twig pattern matching is returned as a set of node tuples, as illustrated on the right side of Fig. 3b.

In addition to twig pattern matching, which is essential for *querying* XML data, another important problem in XML query processing is XML document *filtering*, which arises mainly in applications of selective dissemination of information (SDI) [4]. A core component of SDI is a *document filter*, which matches each incoming XML document D from publishers with a collection of twig queries from subscribers, to determine which subscribed queries have at least one match in D rather than finding all matches of the subscribed queries in D, as required by twig pattern matching. Then, D is sent to the subscribers of the matched queries. In most SDI applications, D arrives in the form of *data streams*, that is, in its original sequential document format, without any associated secondary data structures such as indexes.

Due to the paramount importance of twig pattern matching in XML query processing, in this survey, we focus on reviewing the major techniques for twig pattern matching. More specifically, we focus on twig pattern matching over persistently stored (that is, nonstreaming) XML data.[3]

An intuitive lightweight way of querying persistently stored XML data is based on a *main-memory-style* implementation, which first loads the entire XML document from secondary memory into main memory in the form of a tree and then performs XML queries over this tree. Such main-memory-style XML query processors include Galax [37], XMLTaskForce [45], [47], [48], Saxon [70], and so forth. Although such main-memory-style implementation is straightforward and feasible for querying small XML documents, it is typically inefficient for querying large

3. We give a brief review of techniques for filtering/querying streaming XML data in Section 4.3, in which we introduce the Navigational approach essential for processing streaming XML data.

XML data repositories, since loading the entire XML documents from secondary memory into main memory, regardless of the fact that most persistently stored XML data might be query irrelevant, could cause unnecessarily high disk I/O cost. For achieving high disk I/O performance, which is also one of the core goals of traditional relational database systems, XML data querying and management need to evolve from the lightweight main-memory-style level to a more sophisticated *database-style* level.

Motivated by this, in the recent years, significant effort has been devoted to developing high-performance XML database systems, in which value indexes and structural indexes (see Section 2) could be prebuilt on XML data to improve query processing performance significantly. Specifically, we categorize major techniques into two classes: the *relational approach* (Section 3) and the *native approach* (Section 4). The relational approach directly utilizes existing relational database systems to store and query XML data, whereas in the native approach, specialized storage and query processing systems tailored for XML data are developed from scratch to further improve XML query performance.

Note that many research papers on XML query processing assume that queries work on the tree-shaped XML data model. The first reason for this is that the assumption of the more general graph-shaped data model increases significantly the complexity of XML query processing. The second reason is that graph-shaped XML documents with ID/IDREF attributes are not as common in practical applications as tree-shaped documents. In this survey, we take the assumption of the tree-shaped XML data model, unless noted otherwise (for example, in Section 2.2).

2 Structural Indexes on XML Data

Generally, indexes prebuilt on XML data can facilitate XML query processing by locating goal data quickly while avoiding exhaustive scans of all the data. XML index types include *value indexes* (for example, classical B+-tree indexes), which index data values in XML documents, and *structural indexes*, which index the structure of XML documents. In this section, we review two classes of important structural indexes—numbering schemes and index graph schemes—which have been used in a number of XML query processing techniques.

2.1 Numbering Schemes

An important problem in twig pattern matching is determining structural relationships or, more specifically, reachability between any two nodes in a data tree. For example, to determine whether a pair of A-tagged and B-tagged nodes in a data tree, say, (a, b), matches a path pattern "$A//B$," we need to determine whether there exists a path from a to b in the data tree.

A straightforward method for determining reachability is *tree navigation* (see Lore [81], [82]), which consists of either traversing down the subtree rooted at an A node to see if any B node can be found (*forward navigation*) or backtracking from a B node upward to see if any A node can be found (*backward navigation*). Backward navigation is more efficient than forward navigation when B nodes are more selective (for example, see Fig. 16b). However, the navigation method is, in general, not very efficient, since it may involve traversing a large number of query-irrelevant nodes, that is, nodes tagged with neither A nor B. As we shall see in Section 4.3, such query-irrelevant nodes may scatter across a number of disk pages, accessing which may cause high disk I/O costs.

Another approach to determining reachability is to precompute transitive closures of data trees (XParent [64]). However, the sizes of the transitive closures are typically too large to be used in practice. Thus, it would be desirable to have a method to compactly represent reachability between pairs of nodes in a data tree. One such method is *numbering schemes*.

The work of Dietz [36] is the original work on numbering schemes for trees. It proposed a numbering scheme that we will call $PrePost\ coding$. It labels each node in a tree with a pair of numbers, $(pre, post)$, which correspond to the preorder and postorder traversal numbers of the node in the tree. Zhang et al. [124] introduced PrePost coding into XML applications: it labels each node in an XML data tree with a pair of numbers $(start, end)$, which imply the position of the opening tag $(\langle \ldots \rangle)$ and the closing tag $(\langle / \ldots \rangle)$ of the corresponding element of the node in the XML document. It is easy to see that $(start, end)$ and $(pre, post)$ are essentially the same. We illustrate PrePost coding in Fig. 2a, where each node is adorned with its $(start, end)$ numbers. As illustrated in Fig. 2a, the following property always holds.

Property 1: Ancestor-descendant relationship. *Node a is an ancestor of node b in a data tree if and only if*

$$a.start < b.start < a.end.$$

In PrePost coding, 1) storing $(start, end)$ pairs requires only modest storage space and 2) with the help of $(start, end)$ pairs, we can determine the ancestor-descendant relationship between any two nodes in constant time by using only two number-comparison operations. In addition, we can test for the parent-child relationship by extending $(start, end)$ numbers for each node with another number $level$, which is the depth of the node in the tree.

Property 2: Parent-child relationship. *In a data tree, node a is the parent of node b if and only if $a.start < b.start < a.end$, and $a.level + 1 = b.level$.*

In addition to commonly used "/" and "//" axes, PrePost coding can be used to process all other axes defined in XPath (Section 1.2.1) by adding a $parent$ number to the $(start, end)$ pair for each node, which denotes the parent ID for the node [50], [52], [53].

Note that PrePost coding is also called *interval coding*, since it labels each node with a pair of numbers, which could be viewed as an interval, and determines the reachability between two nodes through checking the containment relationship between their intervals (Property 1). Other interval codings for trees include IndexPost coding [2] and PreSize coding [74].

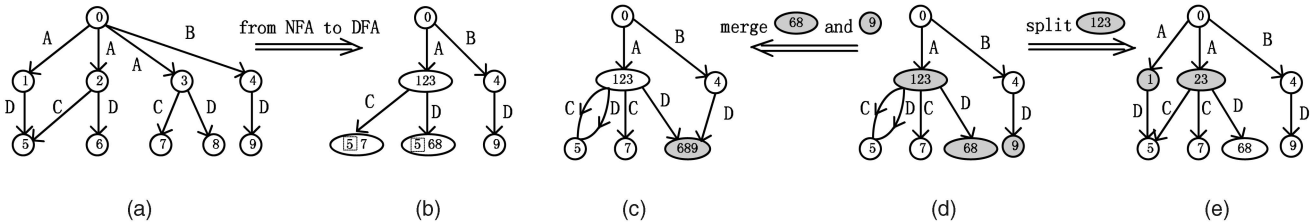


Fig. 4. From data graph to index graph: An example. (a) Data graph (NFA). (b) Strong DataGuide (DFA). (c) A(1)-index. (d) 1-index (A(2)-index). (e) F&B index.

Another well-known numbering scheme for trees is *Dewey coding* [10], which was originally developed for general knowledge classification. Tatarinov et al. [108] introduced Dewey coding into XML query processing. In this approach, each node is associated with a vector of numbers that represents the node-ID path from the root to the node. In Fig. 2a, the Dewey vectors of node 2 and node 10 are 1.2 and 1.2.7.10, respectively. Node a is an ancestor of node b in a data tree if and only if $a.vector$ is a prefix of $b.vector$.

An advantage of Dewey coding over PrePost coding is that Dewey is easier to maintain under dynamic updates on XML data trees. IBM System RX [7],[4] Microsoft SQL Server [88], and Oracle DB [87] have used Dewey coding in their XML query processing components: 1) intuitively, when a new node (or a new subtree) is inserted into a data tree, only the nodes in the subtrees rooted at the *following sibling* nodes of the new node need to update their Dewey vectors. Furthermore, *ORDPATH coding*, a simple variant of Dewey integrated into the Microsoft SQL Server 2005 [88], does not need to update any ORDPATH vectors under insertions. 2) Maintaining PrePost codes is not as straightforward. When a new node (or a new subtree) is inserted into a data tree, all nodes except the nodes in the subtrees rooted at the *previous sibling* nodes of the new node need to update their $(start, end)$ numbers. To reduce this update cost, more sophisticated data structures such as $W\text{-}BOX$ and $B\text{-}BOX$ proposed by Silberstein et al. [106] have to be built for efficiently managing $(start, end)$ numbers.

On the other hand, PrePost coding has its own advantages over Dewey: 1) $(start, end)$ pairs require much less storage space than Dewey vectors and 2) PrePost provides more efficient support for checking reachability between two nodes, as the number-comparison operations are cheaper than checking the prefix-containment relationship between two Dewey vectors. Due to these properties, PrePost coding has been widely used in research projects in XML query processing. Most approaches reviewed in Sections 3 and 4.1 use PrePost coding as their numbering schemes.

Other ad hoc numbering schemes for trees include PBiTree coding [115] and Balanced Index-based numbering scheme for Reconstruction and Decision (BIRD) coding [118], which require arithmetic operations (for example, division) to determine relationships between data nodes.

In addition, numbering schemes have also been developed for DAGs and for more general graphs with cycles,[5] which include tree-based coding [2], [57], [111], 2-hop coding [24], [100], [101], and so forth. Tree-based coding, originally proposed by Agrawal et al. [2], first finds an "optimal" spanning tree in a given DAG and labels that tree by using existing numbering schemes for trees and then assigns additional labels to nodes to keep track of the remaining nontree edges in the DAG. 2-hop coding, originally proposed by Cohen et al. [24], labels each node with a set of its ancestor nodes L_{in} and a set of its descendant nodes L_{out}. Schenkel et al. [100], [101] introduced 2-hop codings into XML applications. Generally, numbering schemes for graphs incur much higher storage space cost and (reachability) query time cost than numbering schemes for trees.

2.2 Index Graph Schemes

Index graph schemes, also called "structural summaries" in some research papers, is another class of important structural indexes. Its basic idea is compressing an XML data graph G into a (smaller) index graph G_I so that XML queries can be more efficiently evaluated over G_I rather than over G. Unlike numbering schemes, index graph schemes are generally applicable to general XML data graphs. Here, we categorize existing index graph schemes into two classes—P-indexes and T-indexes—which are able to cover (linear) path queries and twig queries, respectively.

2.2.1 Index Graphs Covering Path Queries (P-Indexes)

Strong DataGuide, proposed by Goldman and Widom [44], is an early index graph scheme that summarizes all path information in G into G_I. G_I can be viewed as a deterministic finite automaton converted from G, which, in turn, can be viewed as a nondeterministic finite automaton. Note that a data node in G, for example, node 5 in Fig. 4a, might appear within more than one index node in G_I (see Fig. 4b). In general, in the worst case, the size of G_I using Strong DataGuides might be exponential in the size of G. However, Milo and Suciu [84] showed that when G is a tree, Strong DataGuide reduces to 1-index (to be discussed shortly), whose size does not exceed the size of G. Weigel et al. [117] further extended Strong DataGuides to Content-Aware DataGuides to efficiently process DB+IR-style XML queries (see Section 1.2.2) by enhancing DataGuides with value

4. System RX is an experimental prototype that was implemented as an extension to IBM DB2 UDB, and its key techniques have recently been integrated into IBM DB2 9 [59].

5. Note that as He et al. [57] observed, the general graph reachability problem can be reduced to the DAG reachability problem through identifying all *strongly connected components* of a graph.

indexes (inverted indexes [99]) prebuilt on data values in XML documents.

In contrast to Strong DataGuides, many other index graph schemes require each data node to map into exactly one index node. Specifically, they partition data nodes in G into equivalence classes by using some partition strategy and then collect all data nodes in each equivalence class into an index node. Finally, they create a "tag edge" between each pair of index nodes (I_A, I_B) only if there exist two data nodes $a \in I_A$ and $b \in I_B$ connected by one such tag edge in G. In these procedures, no data node is mapped into more than one index node. Therefore, the size of G_I never exceeds the size of G. As a result, incoming XML queries can be efficiently evaluated over G_I to obtain *safe* or *precise* query results. Here, "safe" means that the set of data nodes within index nodes derived by a query over G_I is a superset of the set of real answer nodes, that is, the data nodes derived by the query directly over G, whereas "precise" means that the former nodes are exactly the latter nodes.

Each index graph scheme uses its own data-node partition strategy. Milo and Suciu [84] introduced *1-index*, which partitions data nodes into equivalence classes based on their *B-bisimilarity* (backward bisimilarity [92]): If two data nodes are B-bisimilar, then they have the same set of root paths (a *root path* of a node is a tag path from the root to this node; see Fig. 4d for an example). Note that when G is a tree, its 1-index is also a tree. For example, consider a tree-shaped G, as shown in Fig. 4a, but with edge (2, 5) removed. A 1-index for G will be a tree similar to that shown in Fig. 4d, with {5} merged with {6, 8}. Such a tree-shaped 1-index can also be viewed as a trie if we take the viewpoint of Index Fabric, an index proposed by Cooper et al. [31] for tree-shaped XML data graphs. Index Fabric views the root path of each leaf node in a tree-shaped G as a string of characters, each character corresponding to an edge tag, and then indexes all these strings into a trie G_I. However, unlike the 1-index, which is basically a main-memory data structure, Index Fabric further balances and optimizes the resulting trie for efficient disk access. It has been shown that 1-indexes are precise for all path queries. Unfortunately, although the size of G_I using 1-indexes is at most the size of its source graph G, in many cases, G_I is still too large to be efficiently used in practice.

To further reduce the size of a 1-index, Kaushik et al. [69] generalized 1-index to the $A(k)$-$index$, which partitions data nodes into equivalence classes based on their k-$bisimilarity$: If two data nodes are k-bisimilar, then they have the same set of incoming k-$paths$, where k-path is a tag path of length $\leq k$. 1-index is a special case of A(k)-index when k becomes large enough. An important advantage of the A(k)-index over the 1-index is that A(k)-index generally has a smaller size than 1-index, since A(k)-index puts more data nodes into the same index node. Smaller values of k result in a smaller index size. However, A(k)-index gains this advantage at the expense of query accuracy: it is precise only for those p-path queries whose $p \leq k$. For p'-path queries whose $p' > k$, it is safe but not necessarily precise. Fig. 4c shows an A(1)-index. Note that the 1-index in Fig. 4d is also an A(2)-index. Two index nodes {6, 8} and {9} in Fig. 4d are merged into one index node {6, 8, 9} in Fig. 4c, since nodes 6, 8, and 9 have the same set of incoming 1-paths $\{D\}$. Although 1-path queries such as "$/A$" and "$//D$" are precise over this A(1)-index, some 2-path queries such as "$/A/D$" and "$/B/D$" are only safe but not precise. For example, evaluating "$/A/D$" over this A(1)-index will return index nodes {5} and {6, 8, 9}, whereas node 9 in G does not match "$/A/D$." Therefore, for p'-path queries, an extra *postvalidation* step is required when precise query results are called for. This step validates the data nodes derived from G_I via rechecking their incoming paths in G and results in extra computation time. Thus, in practice, it is very important to select an appropriate value of k. Although a k that is too small lowers the performance of long-path queries, a k that is too large may result in significantly larger indexes and thus may increase the evaluation costs of both short- and long-path queries.

Chen et al. [13] further generalized A(k)-index to an adaptive $D(k)$-$index$, which assigns different k values (rather than the same k value, as in A(k)-index) to different index nodes based on a specific query workload. In particular, it assigns small k values to tag names that usually appear in short-path queries and large k values to tag names that usually appear in long-path queries, which makes the resulting D(k)-index both (index) space- and (query) time-efficient for the given query workload. Further, two procedures, "promotion" and "demotion," were developed in [13] to periodically tune the k values of index nodes to make D(k)-index adaptive to incrementally changing query workloads. *Adaptive Path indEX for XML data (APEX)*, proposed by Chung et al. [23], is another adaptive index graph scheme, whose motivation and function are similar to those of D(k)-index. In particular, a *hash tree* index H_{APEX} was developed in [23] to efficiently retrieve (answer) index nodes from the *APEX* index graph when path queries are given. A further improvement of D(k)-index is due to the work of He and Yang [58], in which two sophisticated indexes, M(k)-index and $\mathrm{M}^*(\mathrm{k})$-index, were developed to avoid "overrefinement," which creates for some index nodes unnecessarily large k values that are irrelevant to frequent queries.

2.2.2 Index Graphs Covering Twig Queries (T-Indexes)

P-indexes cover (linear) path queries only. Answering general twig queries requires an additional *path-joining* step on the data nodes returned by path queries over the P-index graph, as discussed in Section 4.1.4. For example, ToXin [98] uses DataGuide as its path index and then implements path joining by using the Edge approach (see Section 3.1). To avoid such path joining, some projects have focused on developing T-indexes, which can directly cover general twig queries while generally having larger index size than P-indexes.

The $F\&B$-index, proposed by Abiteboul et al. [1], partitions data nodes into equivalence classes based on their $F\&B$-$bisimilarity$ (forward and backward bisimilarity [92]). Unlike B-bisimilarity in 1-index, F&B-bisimilarity depends not only on incoming (backward) paths but also on outgoing (forward) paths of data nodes. Thus, {1, 2, 3} in Fig. 4d is split into two index nodes, {1} and {2, 3}, in Fig. 4e, since the outgoing paths of node 1 are different from those of nodes 2 and 3 (see Fig. 4a). As a result, the precise query

Label	Source	Target	Flag	Value
publisher	1	2	Element	null
name	2	3	Attribute	MIT Press
address	2	4	Value	Cambridge
publisher	1	9	Element	null
...	...	...	...	...

(a)

Label	Start	End	Level	Flag	Value
publisher	2	20	1	Element	null
name	3	5	2	Attribute	MIT Press
address	6	8	2	Value	Cambridge
publisher	21	38	1	Element	null
...	...	...	...	...	...

(b)

Fig. 5. Examples of (a) an edge table and (b) a node table.

answer {2, 3} to the twig query "$/A[/C \text{ AND } /D]$" can be obtained directly from the F&B-index in Fig. 4e but not from the 1-index in Fig. 4d. Kaushik et al. [67] showed that the $F\&B$-index is the smallest index that can cover all twig queries. Unfortunately, the size of the F&B-index is usually unacceptably large. Motivated by this, Kaushik et al. [67] developed a flexible $(\mathrm{F}+\mathrm{B})^i$-index that can cover only a subclass of twig queries but has a smaller index size than the F&B-index. The value of i is used for tuning the trade-off between index size and query answering power. With a smaller i value, $(\mathrm{F}+\mathrm{B})^i$-index has a smaller size but can only cover a smaller subclass of twig queries. Theoretical results by Ramanan [96] show that when twig queries do not include the NOT operator, an index built by partitioning data nodes based on F&B-similarity [83] rather than on F&B-bisimilarity is the smallest index that can cover such twig queries. It is never larger (and is exponentially smaller in some cases) than the F&B-index. Recently, Wang et al. [116] developed a disk-based F&B-index to efficiently deal with those cases where the F&B-index is too large to fit in memory.

2.2.3 Summary on Index Graph Schemes

From the above discussion, we can see that index graph schemes have evolved from P-indexes covering only path queries to larger-sized T-indexes covering general twig queries. Further, both P-indexes and T-indexes have undergone the development from precise indexes with potentially large sizes such as 1-index in P-indexes and $F\&B$-index in T-indexes to safe-only indexes with smaller sizes such as A(k), D(k), M(k), and APEX indexes in P-indexes and $(\mathrm{F}+\mathrm{B})^i$-index in T-indexes. No single index graph scheme dominates the others, since there is always a trade-off between index size and query answering power. As observed earlier for A(k)-index, although safe-only indexes have smaller index sizes, they require an extra expensive postvalidation step to obtain precise query results when queries are beyond their covering range.

2.3 Summary

Numbering schemes and index graph schemes are not just two independent and exclusive tools for indexing the structure of XML data. In Section 4.1.4, we will see that they play different roles in answering XML queries: index graph schemes are used for path selection, and numbering schemes are for path joining. They could be used together to improve query processing performance.

3 XML Query Processing: The Relational Approach

Many of today's commercial database systems are relational database management systems (RDBMSs), and examples include IBM DB2, Microsoft SQL Server, and Oracle DB. For more than 30 years of academic and industrial efforts, RDBMSs have acquired strong capabilities in storage management, query processing and optimization, and concurrency control and recovery. Motivated by this fact, a number of research projects have addressed storing and querying XML data in RDBMSs.

3.1 The Edge Approach

3.1.1 The Basic Edge Approach

Florescu and Kossmann [41] proposed a simple approach to shredding XML data into relations. This approach places all edges in an edge-labeled XML data tree into a single relational table $Edge$, whose schema is shown in Fig. 5a (the table in Fig. 5a is populated with the XML data of Fig. 2b). The key idea here is using an attribute pair $(Source, Target)$, which represents the two end points of each edge. $Label$ represents the tag on an edge, whereas $Flag$ and $Value$ give the type and value, respectively, of the target node of an edge.

Two edges A and B can be joined together if and only if $A.Target = B.Source$. Based on this property, it is easy to transform XML queries without "//" axes into SQL queries, as illustrated in Fig. 6a. Evaluation of such SQL queries comprises two main steps. The first step is *edge selection* (part 1), which retrieves the data edges for each label in the query. A clustered index prebuilt on $Label$ can significantly speed up processing in this step. In addition, an index prebuilt on the $Value$ attribute can help efficiently retrieve some data edges, for example, the $address$ edges with the "Cambridge" value, as shown in Fig. 6. The second step is *edge joining* (part 2), which joins adjacent data edges retrieved in part 1. This step can be done in a more efficient way by using prebuilt indexes on $(Source, Target)$.

3.1.2 The Binary Approach

Executing part 1 of the basic Edge approach can be avoided by using the *Binary* approach proposed by Florescu and Kossmann [41], which pregroups all edges in the Edge table by their $Labels$ and creates one table for each distinct $Label$. Each table has the schema $(Source, Target, Flag, Value)$, with $Label$ dropped from the Edge schema. An example is shown in Fig. 6b, in which the edge-selection operations of part 1 of Fig. 6a are not performed. In addition to improving query performance, the Binary approach also saves storage space, since the $Label$ attribute is not stored.

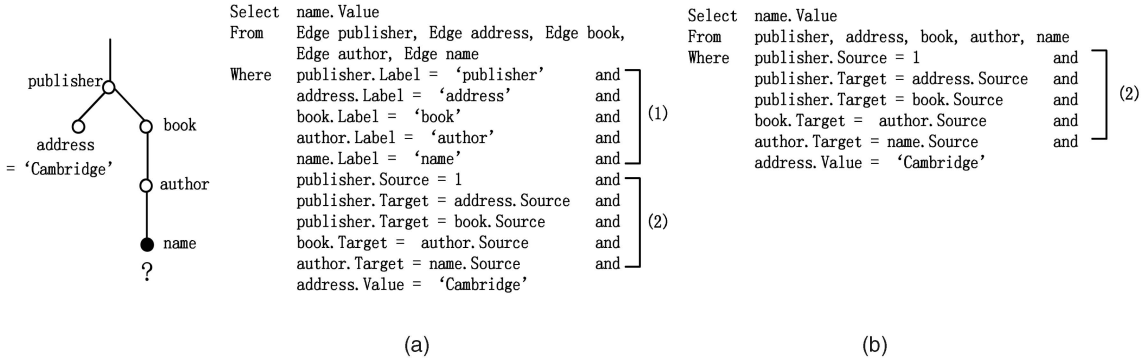


Fig. 6. The Edge approach: SQL query for "$/publisher[address = \text{“}Cambridge\text{”}]/book/author/name$." (a) The basic edge approach. (b) The binary approach.

The Edge approach has the following limitations: 1) It involves a number of join operations. The number of joins is the number of query nodes in a twig query (minus 1). Thus, the approach may fail to process large twig queries efficiently. 2) It may fail to process queries with "//" axes (for example, "$A//B$") efficiently because it is hard to determine the number or names of the tags between A and B. For determining the ancestor-descendant relationships between data nodes, it needs to partially compute transitive closures of data trees by issuing expensive recursive SQL queries [40]. Precomputing and materializing transitive closures (XParent [64]) can avoid such expensive computation but may incur very high storage-space costs.

3.2 The Node Approach

As introduced in Section 2.1, a class of important structural indexes called numbering schemes may help answer "//"-axis queries efficiently. Zhang et al. [124] developed a $Node$ approach, in which all internal nodes (that is, the element and attribute nodes) in a node-labeled XML data tree are stored in a relational table $Node$, whose schema is shown in Fig. 5b and which is populated with the XML data in Fig. 2a. The key idea here is using the attribute triple $(Start, End, Level)$. "//"-axis queries can be answered efficiently by using the $(Start, End)$ pairs. $Level$ is used along with $(Start, End)$ to answer "/"-axis queries.

Based on Properties 1 and 2 in Section 2.1, it is easy to transform queries with both "/" and "//" axes into SQL queries, as illustrated in Fig. 7. Similar to the Edge approach, evaluation of the SQL queries consists of two steps: node selection (part 1) and node joining (part 2). Part 2 joins the data nodes retrieved by part 1 via their $(Start, End, Level)$ numbers. Just as in the Edge approach, executing part 1 can be avoided in the Node approach by using a modification similar to that in the Binary approach.

Unlike the Edge approach, the Node approach does support "//"-axis queries efficiently. However, similar to the Edge approach, it may involve a number of join operations, which may impact the processing performance of large twig queries. Specifically, the number of required joins is the number of query nodes in a twig query (minus 1).

3.3 The Path Materialization (PM) Approach

3.3.1 The Basic PM Approach

To reduce the number of node joins, Yoshikawa et al. [123] proposed a PM approach, in which internal nodes in a node-labeled XML data tree are stored in a relational table $Path$, whose schema is shown in Fig. 9a and which is populated with the XML data in Fig. 2a. The $Path$ table is very similar to the $Node$ table. The difference is that rather than storing the tag of each node in the $Label$ attribute, the PM approach stores the tag path from the root to each node (called *root path*) in a $Path$ attribute.

Using the $Path$ attribute, the PM approach can answer twig queries efficiently in units of paths rather than in units of edges. Specifically, given a twig query, the PM approach first decomposes it into multiple root-to-leaf path queries and then joins the results of the path queries, as illustrated in Fig. 8. Evaluation of the SQL queries consists of two main steps: path selection (part 1) and path joining (part 2). Part 1 uses the root paths of the leaf nodes ($address$ and $name$) and of the branching nodes ($publisher$) in the query twig to retrieve the corresponding data nodes from the data tree.

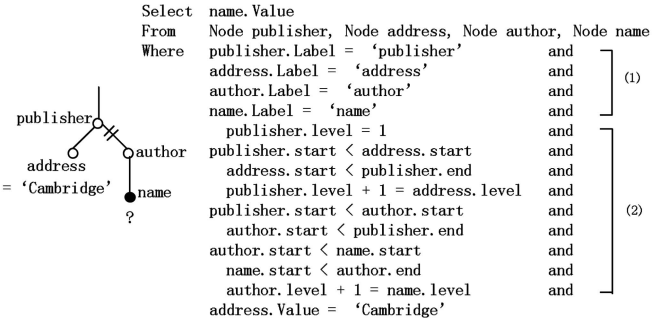


Fig. 7. The Node approach: SQL query for "*/publisher[address = "Cambridge"]//author/name*."

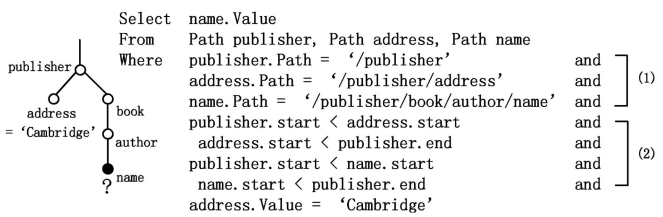


Fig. 8. The basic PM approach: SQL query for "*/publisher[address = "Cambridge"]/book/author/name*."

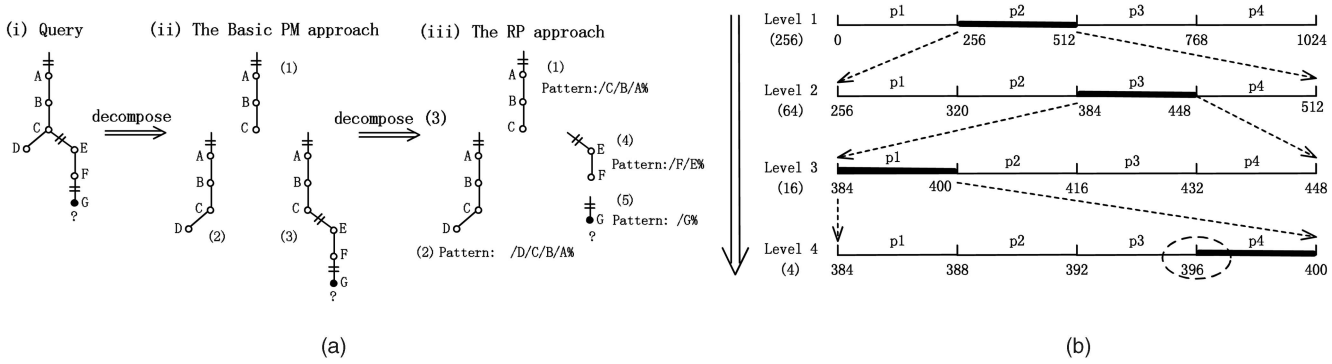
Path	Start	End	Level	Flag	Value
/publisher	2	20	1	Element	null
/publisher/@name	3	5	2	Attribute	MIT Press
/publisher/address	6	8	2	Value	Cambridge
/publisher	21	38	1	Element	null
...	...	...	...	...	...

(a)

ReversedPath	ORDPATH	Flag	Value
/publisher	1.1	Element	null
/@name/publisher	1.1.1	Attribute	MIT Press
/address/publisher	1.1.3	Value	Cambridge
/publisher	1.3	Element	null
...	...	...	...

(b)

Fig. 9. Examples of (a) a Path table and (b) a ReversedPath table.



(a) (b)

Fig. 10. (a) The RP approach and (b) the BLAS approach: PLabel $(''/\mathrm{p2/p3/p1/p4}'') = 396$.

Part 2 joins the data nodes retrieved by part 1 via their $(Start, End, Level)$ numbers.

We note two features of the PM approach: 1) It involves fewer join operations in part 2 than the Node approach because PM answers twig queries in units of paths rather than in units of edges. In the example in Fig. 8, the Node approach would need to join five query nodes, whereas the PM approach needs to join only three query nodes. 2) Similar to the Node approach, the PM approach can support "//"-axis queries by using the Optional String Pattern Matching (OSPM) function ("LIKE") provided by SQL. For example, to answer query "$/publisher//name$," we can use "name.Path LIKE "$/publisher\%name$."".

Besides XRel [123], XParent [64] and MonetDB [102] also employ PM-like approaches, which have the same implementation for part 1 but differ in the implementation of part 2. XParent [64] designs a Path + Edge table, which replaces $(Start, End, Level)$ in Fig. 9a with $(Source, Target)$, as in the Edge approach. To efficiently implement part 2 when "//" axes are involved, all $(descendant, ancestor)$ node pairs in the data tree are precomputed and stored in a separate $Ancestor$ table, which may result in high storage-space costs. MonetDB [102] pregroups all edges in the Path + Edge table by their $Paths$ and creates one table for each distinct $Path$ (rather than one table for each distinct $Label$, as in the Binary approach). Without the materialized $Ancestor$ table as in XParent, its implementation for part 2 has limitations similar to those of the Edge approach when processing "//" axes.

Although the PM approach reduces the number of joins in part 2, it does so at the expense of increasing the complexity of the selection operation in part 1. It is known that SQL can support exact string pattern matching ("=") efficiently through prebuilt B+-tree indexes on strings. However, B+-tree indexes cannot support "LIKE" efficiently (to find patterns with multiple "%"'s, a large number of irrelevant strings might have to be exhaustively scanned). Therefore, PM may not support efficiently queries with multiple "//" axes.

Another limitation of PM is that it might result in incorrect query answers when recursion (Section 1.1) exists in XML data. Consider a path query "$//A[/B//C]$" over a recursive data path "$a_1{-}b_1{-}a_2{-}c_1$." Although only a_1 should be in the query result, PM returns both a_1 and a_2, since path selection using "$\%A$" returns $\{a_1, a_2\}$, path selection using "$\%A/B\%C$" returns $\{c_1\}$, and path joining between $\{a_1, a_2\}$ and $\{c_1\}$ returns $\{a_1, a_2\}$.

3.3.2 The Reversed-Path (RP) Approach

Pal et al. [90] proposed an RP approach, which overcomes the main drawback of the PM approach. The RP approach uses the relation schema shown in Fig. 9b. The key idea of RP is storing *reversed* root paths of data nodes in a $ReversedPath$ attribute. In addition, RP in [90] uses an $ORDPATH$ attribute instead of the $(Start, End, Level)$ attributes of the PM approach. $ORDPATH$ coding [88] is a variant of the Dewey coding (see Section 2.1). Similar to PrePost coding, it can be used to determine reachability between nodes. In the discussion below, we focus on the $ReversedPath$ attribute.

Fig. 10a illustrates how the RP approach answers twig queries with multiple "//" axes. The first step is path selection: The query twig is first decomposed into three paths, as in the basic PM approach. Path 3 involves three "//" axes. The basic PM uses "$\%A/B/C\%E/F\%G$" as a search pattern on the $Path$ attribute to retrieve the corresponding data nodes. As discussed earlier, it is not straightforward to implement efficiently this type of pattern matching. The RP approach further decomposes path 3 into path 4 and path 5, each of which includes only one "//" axis and only in the beginning. Thus, we can

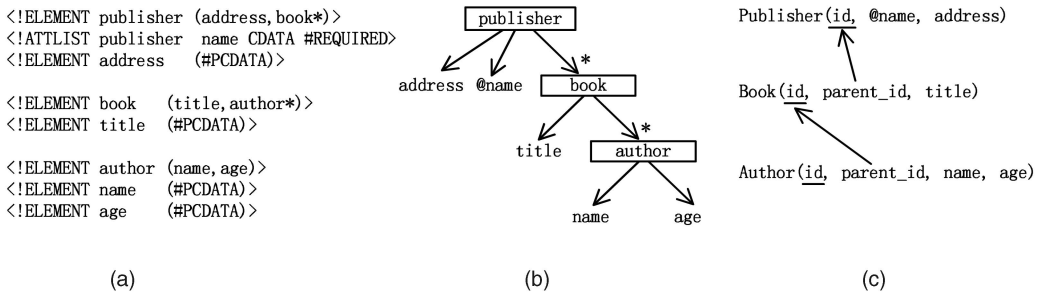


Fig. 11. A DTD and its relational schema. (a) A DTD document. (b) A DTD tree. (c) Relational schema.

use "$/F/E\%$" and "$/G\%$" as search patterns on the $ReversedPath$ attribute. The goal here is to find strings with a specified prefix, which can be implemented more efficiently than the "LIKE" matching with multiple "%" symbols. In the final path-joining step, the RP approach joins the results of the path queries by using the $ORDPATH$ attribute.

The RP approach always returns correct query results, even in the presence of recursion in XML data, since each path generated by its path-decomposition procedure includes only one "//" axis and only in the beginning. Consider the example that we used in the basic PM approach, in which a path query "$//A[/B//C]$" is issued over a recursive data path "$a_1{-}b_1{-}a_2{-}c_1$." RP returns the correct query result $\{a_1\}$ because it uses three path selections, "$/A\%$," "$/B/A\%$," and "$/C\%$," rather than only two path selections, "$\%A$" and "$\%A/B\%C$," as in the basic PM approach.

The RP approach has been integrated into IBM System RX [7], Microsoft SQL Server 2005 [88], [90], and Oracle DB [87] and was also independently proposed by Chen et al. [19].

3.3.3 The BLAS Approach

As we have seen, the RP approach reduces general-purpose "LIKE" matching to the easier task of string prefix matching (SPM). However, [7], [19], [87], and [90] do not discuss efficient implementations of SPM. It seems that SPM is simply pushed down to the SQL engine. Chen et al. [16], in addition to discovering RP independently, proposed a sophisticated BLAS approach to implementing SPM efficiently. The schema of the BLAS table is given as follows:

PLabel	Start	End	Level	Flag	Value

The key idea of BLAS is encoding each ReversedPath string into a $PLabel$ value. This encoding is a variant of the reversed arithmetic encoding developed by Min et al. (XPRESS [85]), and we give an illustration in Fig. 10b. In the example, we assume that our XML document has a total of four distinct tag names, p_1 through p_4. At the first level, these four tags divide the reserved number space [0, 1,024) into four equal-length segments, each with length $1,024/4 = 256$. Similarly, at the second level, four tags divide each first-level segment into four equal-length segments of length $256/4 = 64$ each, and so on. Then, we have

$$\begin{aligned} &PLabel(\,''/p_2/p_3/p_1/p_4'') \\ &= 256 * (2-1) + 64 * (3-1) + 16 * (1-1) + 4 * (4-1) \\ &= 396. \end{aligned}$$

A nice property of PLabel is that all strings with common prefixes are clustered in adjacent positions in the $PLabel$ number space. Thus, all ReversedPath strings with a specified prefix can be retrieved efficiently through an SQL range query if a clustered B+-tree index has been prebuilt on the PLabel attribute in the $BLAS$ table. For example, to retrieve all reversed paths with prefix "$/p_2/p_3/$," BLAS computes $lower\_bound(\,''/p_2/p_3/'') = PLabel(\,''/p_2/p_3/'') = 384$ and $upper\_bound(\,''/p_2/p_3/'') = PLabel(\,''/p_2/p_4/'') = 448$ and then issues an SQL range query to retrieve all reversed paths with PLabel within [384, 448). Fig. 12 illustrates this idea. Path selection in part 1 is implemented efficiently using SQL range queries, whereas part 2 joins the data nodes retrieved in part 1 through their $(Start, End, Level)$ numbers.

3.4 The DTD Approach

All the above approaches address storing and querying general (schemaless) XML data. In many practical applications, XML data also conform to a "schema" to some extent because a common agreement on the schemas of the data would facilitate significantly the data exchange among various applications. Such schema information provides extra opportunities for more compact storage and more efficient querying of XML data [109]. The DTD approach that we review in this section utilizes such crucial schema information.

XML schemas can be described using DTDs [26], [21] or XML-Schemas [29], which are essentially extensions to DTDs. We now briefly introduce basic issues of DTDs only. A DTD is a set of statements that specify 1) relationships between XML elements and their subelements or attributes and 2) the data type of XML elements or attributes. Fig. 11a gives an example of a DTD document, whose semantics can be explained using the DTD tree in Fig. 11b. The "$*$" symbol

```
Select G.Value
From   BLAS C, BLAS D, BLAS F, BLAS G
Where  lower_bound( '/C/B/A' )   <= C.PLabel < higher_bound( '/C/B/A' )   and  ┐
       lower_bound( '/D/C/B/A' )<= D.PLabel < higher_bound( '/D/C/B/A' ) and  │
       lower_bound( '/F/E' )     <= F.PLabel < higher_bound( '/F/E' )     and  │ (1)
       lower_bound( '/G' )       <= G.PLabel < higher_bound( '/G' )       and  ┘
       C.start < D.start   and    D.start < C.end                              and  ┐
         C.level + 1 = D.level                                                  and  │ (2)
       C.start < F.start   and    F.start < C.end                              and  │
       F.start < G.start   and    G.start < F.end                                   ┘
```

Fig. 12. The BLAS approach: SQL for the twig query in Fig. 10a.

```
Select Author.name
From   Publisher, Book, Author
Where  Publisher.id = Book.parent_id    and
       Book.id = Author.parent_id       and
       Publisher.address = 'Cambridge'
```
(2)

Fig. 13. The DTD approach: SQL query for "*/publisher[address "Cambridge"]/book/author/name*" (and also for "*/publisher[address ="Cambridge"]//author/name*").

associated with an element in a DTD implies that multiple copies of the element can be present under its parent element. For example, a $publisher$ element might have multiple $book$ subelements.

Unlike the schemaless approaches (such as Edge, Node, and PM), which generate the same relational schema (tables $Edge$, $Node$, $Path$, and so forth) for all types of XML data, regardless of their structure, the DTD approach generates different relational schemas for different DTDs [104], [105]. Consider the example in Fig. 11c, where relations are created for the root element $(publisher)$ and for all $*$-elements ($book$ and $author$). Each relation has an id attribute as its key, and each $*$-element relation has a $parent\_id$ attribute, which is a foreign-key reference to its parent-element table. Note that $(id, parent\_id)$ in each $*$-element relation represents an edge, similar to $(Source, Target)$ in the Edge table.

The DTD approach transforms XML queries into SQL queries based on the schema information in the DTD tree: 1) For a "/"-axis join A/B, it first checks whether A is the parent of B in the DTD tree. If not, then A/B is an invalid query. Otherwise, relations A and B are joined using $A.id = B.parent\_id$, similar to $A.Target = B.Source$ in the Edge approach. Fig. 13 gives an example. 2) For a "//"-axis join $A//B$, it first checks whether A is an ancestor of B in the DTD tree. If not, then $A//B$ is an invalid query. Otherwise, relations A and B and all relations between them (which can be found in the DTD tree) are joined using the "/"-axis join above. For instance, the SQL query in Fig. 13 also works for "$/publisher[address = "Cambridge"]//author/name$" because the DTD tree in Fig. 11b implies that only $book$ can appear between $publisher$ and $author$. Note that for either A/B or A//B, the join operation between relations A and B can be avoided if B has been inlined into relation A as an attribute rather than being stored independently as a relation, as we shall see shortly.

Compared to the schemaless approaches, the DTD approach could reduce the number of joins significantly, for the following reasons: 1) the non-$*$ elements in DTDs, for example, $address$ and $name$ in Fig. 11, have been inlined into the $*$-element relations as attributes and can be retrieved using the projection or selection operators rather than more expensive joins. For example, for query "$/publisher[address = "Cambridge"]/@name$," an SQL solution is

$$\text{SELECT Publisher.name FROM Publisher}$$
$$\text{WHERE Publisher.address} = \text{"Cambridge."}$$

Note that this SQL solution does not require any joins, whereas the schemaless approaches would require two joins. 2) The DTD approach does not require joins for some existence-test query nodes. For example, for query "$/publisher[address]/book[title = "Databases"]$," an SQL solution is

$$\text{SELECT Book.id FROM Book}$$
$$\text{WHERE Book.title} = \text{"Databases."}$$

That is, $publisher$ does not need to be joined with $address$ or $book$, since the DTD tree implies that each $book$ node must have a $publisher$ parent, which must have an $address$ child.

On the other hand, the DTD approach might involve more joins than the Node or PM approaches, since it transforms each "//"-axis join into a series of "/"-axis joins. However, as noted by Krishnamurthy et al. [72], this inefficiency could be alleviated by augmenting the DTD approach with numbering schemes, as in the Node and PM approaches, that is, by introducing the $(Start, End, Level)$ attributes into relations. For instance, when the $Publisher$ and $Author$ relations in Fig. 11c are augmented this way, answering

$$"/publisher[address = "Cambridge"]//author/name"$$

requires only one join (between the $Publisher$ and $Author$ relations) rather than two joins, as in Fig. 13.

From the above, we can see that by using the schema information in DTDs, the DTD approach could generally have better performance than the schemaless approaches, in particular when the number of "//"-axes in queries is small or when the DTD approach is augmented with numbering schemes.

Note that in the above, we consider only tree-shaped DTDs. In practice, DTDs can be DAGs or even general graphs with cycles, even for tree-shaped XML data: 1) a DAG-shaped DTD graph may be caused by some node in the DTD graph having more than one parent node. For example, a $book$ node in a DTD graph might have not only a $publisher$ parent node, as in the DTD tree in Fig. 11b, but also a $vendor$ parent node; that is, both the $publisher$ and $vendor$ data elements might have $book$ subelements. In such cases, we need to create two $Book$ relations: one storing all $book$ elements whose parents are $publisher$ elements and having its $parent\_id$ attribute refer to the $Publisher$ relation and another storing all $book$ elements whose parents are $vendor$ elements and having its $parent\_id$ attribute refer to the $Vendor$ relation. Note that the $book$ elements stored in these two $Book$ relations are disjoint, since each (nonroot) data element in a tree-shaped XML document has exactly one parent element. For a query "$//vendor/book$," the $Vendor$ relation is joined with the second $Book$ relation only. 2) A DTD graph with cycles may be caused by recursion in XML data. For example, multiple pairs of $(book, author)$ data nodes might be nested on the same path in a data tree. In such cases, the DTD graph has not only an edge from the $book$ node to the $author$ node, as in the DTD tree in Fig. 11b, but also another edge from the $author$ node back to the $book$ node, which forms a cycle in the DTD graph. Due to this backward edge, the $book$ node in Fig. 11b now has two "parents": $publisher$ and $author$. Thus, as in the DAG case, we need to create two $Book$ relations: one storing all $book$ elements whose parents are $publisher$ elements and having its $parent\_id$ attribute refer to the $Publisher$ relation and another storing all $book$ elements

whose parents are $author$ elements and having its $parent\_id$ attribute refer to the $Author$ relation. For a query "$//book[title = "Databases"]//author$," after all $book$ elements with title "Databases" have been selected, a recursive SQL query over these selected $book$ elements, the second $Book$ relation, and the $Author$ relation is needed for computing the $(author)$ transitive closure of these selected $book$ elements, as in the Edge approach, since the axis of $author$ in the query is "//." Using recursive SQL queries can be avoided when the DTD approach is augmented with numbering schemes, as discussed earlier in this section.

A more detailed overview of the state of the art and of open problems of the DTD approach can be found in [72].

3.5 Summary

The relational approach stores XML data in relational databases and transforms twig queries over XML data into SQL queries over relational data. In this approach, all query processing and optimization efforts can be pushed into the relational query optimizer. We note that 1) when XML data are schemaless, the PM approach has advantages over the Edge and Node approaches because PM a) supports "//"-axis queries efficiently and b) may require fewer join operations. Furthermore, among the three versions of PM, BLAS (being an extension of the basic RP approach with PLabeling) appears to be the best in terms of query processing performance. The basic RP approach has been integrated into IBM System RX [7], Microsoft SQL Server 2005 [88], [90], and Oracle DB [87]. 2) Also, when XML data conform to a schema, the DTD approach could generally have better performance than other schemaless approaches.

4 XML Query Processing: The Native Approach

Although the relational approach is simple and straightforward to implement, it may not exhibit optimal query processing performance. To answer "//"-axis queries efficiently, the Node and PM approaches use θ-joins[6] to implement node/path-joining (see part 2 in Figs. 7, 8, and 12) while discarding equijoins used in the Edge approach (see part 2 in Figs. 6 and 13). θ-joins are more complex and expensive than equijoins. Although state-of-the-art RDBMSs have been coupled with efficient techniques for processing equijoins, they typically do not support θ-joins efficiently, in particular when queries involve multiple inequality-comparison predicates. The experimental work by Zhang et al. [124] has verified this point. Motivated by this, many native techniques have been developed to query XML data efficiently. We call these techniques *native approaches*, since their query processing (and, perhaps, also storage) mechanisms are developed from scratch, without involving relational databases. Native approaches are driven by the belief that a storage and query processing system specifically tailored for XML data will improve XML query performance significantly.

6. θ-joins are joins involving "$>$" or "$<$" comparisons, whereas equijoins involve only "$=$"comparisons.

```
If  cursor_B.start < cursor_A.start  Then
   advance cursor_B;
Else
   temp_cursor_B = cursor_B;
   While( temp_cursor_B.start < cursor_A.end )   // the inner-loop join
      Output a tuple solution into join results. Specifically,
          Case 1 (For the `A/B' query):
              Output (cursor_A, temp_cursor_B) if cursor_A.level+1 = temp_cursor_B.level;
          Case 2 (For the `A//B' query):
              Output (cursor_A, temp_cursor_B);
      advance temp_cursor_B;
   Endwhile
   advance cursor_A;
```

Fig. 14. Core of the MPMGJN algorithm.

4.1 The Join Approach

In this section, we review the Join approach, a very important native approach, which implements efficiently θ-joins involved in XML twig queries (such θ-joins are also called "structural joins" in many research papers). In this approach, XML data are stored in inverted lists. The concept of inverted lists originates from inverted indexes, which have been widely used in IR to implement text search efficiently [99]. An inverted list is created for each distinct tag in XML documents, and each list records the positions of all elements with that tag name, where the position of an element is expressed using its $(Start, End, Level)$ numbers (or Dewey vectors [79], [122]). Elements in each list are sorted in the increasing order of their start numbers. The following illustrates inverted lists for Fig. 2a:

<publisher>:	(2, 20, 1)	(21, 38, 1)
<book>:	(9, 19, 2)	(22, 34, 2)
<...>:	...	

We can see that inverted lists here are essentially the same as the $Node$ table in the relational approach (see Fig. 5b), provided that all nodes in the $Node$ table are pregrouped by their $Labels$, as in the Binary approach (Section 3.1). This implies that the Join approach could utilize relational storage for XML data while providing different query processing techniques than existing RDBMSs.

4.1.1 The Multi-Predicate MerGe JoiN (MPMGJN) Approach

Zhang et al. [124] proposed an MPMGJN algorithm, whose implementation is somewhat similar to the classical merge-join algorithm developed in relational query optimizers for equijoins. To answer a query "$A//B$" or "A/B," first, two cursors are created to point to the heads of $list_A$ and $list_B$, respectively. Then, the two cursors are compared with each other and are advanced as needed to implement the merge join. In contrast to the standard merge-join implementation for equijoins, MPMGJN has its own cursor advancing mechanism, which is tailored to efficiently support structural joins. Specifically, at each step, it compares and advances two cursors, as shown in Fig. 14. As illustrated in Figs. 15a and 15b, the inner loop join at a_i corresponds to a range scan $(a_i.start, a_i.end)$ over $list_B$. The experimental work in [124] shows that for many XML queries, MPMGJN is more than an order-of-magnitude faster than current RDBMS join implementations. In XML Indexing and Storage System (XISS) [74], Li and Moon independently developed another merge-join algorithm $\varepsilon\varepsilon$-Join, which is

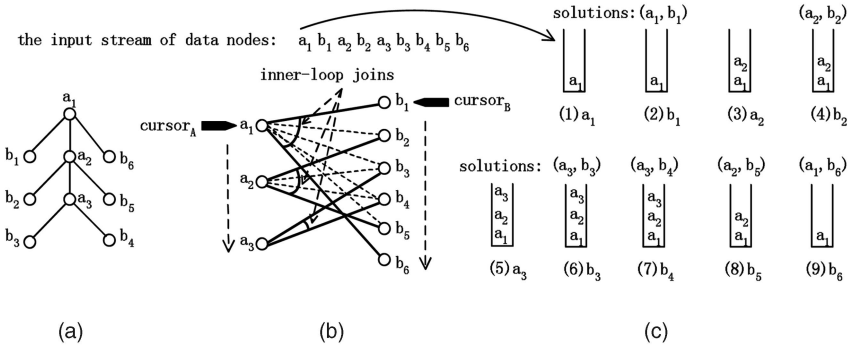


Fig. 15. Applying MPMGJN and StackTree to query "A/B." (a) Data tree. (b) The MPMGJN approach. (c) The StackTree approach.

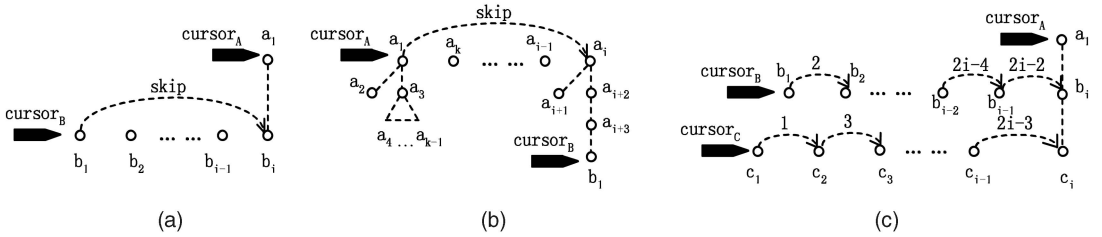


Fig. 16. Skip: no need for sequential reads. (a) $A//B$: skip descendants. (b) $A//B$: skip ancestors. (c) $A//B//C$: fixing the broken query edge (B. C) first.

similar to MPMGJN but uses PreSize coding, another type of interval coding (Section 2.1), rather than PrePost coding.

Zhang et al. [124] observed that the inner loop join at a_i in MPMGJN can also be implemented by issuing a range query $(a_i.start, a_i.end)$ to probe a B+-tree index prebuilt on the $start$ numbers of $list_B$. Such index-probe operation lays a foundation for index nested-loop join (INLJ) algorithms, for example, the INLJ algorithm proposed by Moro et al. [86] and the Staircase Join algorithm proposed by Grust et al. [50], [52], [53]. If the A nodes are very selective (see Fig. 16a), then an INLJ algorithm can outperform MPMGJN, as it does not need to sequentially read $list_B$. However, as reported in the experimental work in [124], the INLJ algorithm may be inferior to MPMGJN in other cases, since it may incur significant random disk I/O costs due to the index-probe operation at each A node.

4.1.2 The StackTree Approach

Al-Khalifa et al. [3] observed that MPMGJN fails to process "/"-axis queries efficiently in some cases. A motivating example is shown in Fig. 15b. In the example, a_1 has only two B children: b_1 and b_6. However, in MPMGJN, the inner loop join at a_1 also visits unnecessarily b_2 through b_5, which are proper descendants but not children of a_1.

Al-Khalifa et al. [3] proposed a StackTree approach that can avoid such visiting of unnecessary nodes. StackTree uses a stack structure to cache those A nodes that are nested on the same path in data trees. Fig. 17 shows the core of the StackTree algorithm. At each step, the data node with the smallest $start$ number is taken out of its list. If it is an A-tagged node, then it is pushed into the stack. If it is a B-tagged node, then StackTree tries to use it to form tuple solutions with A-tagged nodes in the current stack. Fig. 15c illustrates this process, in which b_3 is compared with a_3 only (step 6) rather than with a_1 through a_3, as in Fig. 15b. Generally, StackTree shows better query processing performance than MPMGJN [3].

Join order. Both StackTree and MPMGJN are binary-join algorithms; that is, they join only a pair of inverted lists (that is, only two nodes in the query twig) at a time. Since a complete twig query consists of a series of binary joins, and different join orders result in different sizes of intermediate join results, the join order has a significant impact on XML query processing performance. Similar to the context of relational databases, where many query optimizers use the classical cost-based dynamic-programming method [103] to select an optimal join order, Wu et al. [120] proposed cost-based dynamic-programming methods for selecting an optimal or near-optimal order of binary structural joins for XML twig queries, in which the sizes of intermediate join results are estimated using the histogram techniques proposed in [119]. The StackTree join algorithm and its corresponding join-order-selection algorithms based on dynamic programming have been integrated into Tree-structured native XML database Implemented at the University of Michigan by Bright Energetic Researchers (TIMBER) [60].

Output order. Another important issue is that the StackTree algorithm in Fig. 17 outputs all tuple solutions in the increasing order of start numbers of descendant nodes (that is, of B-tagged nodes). For example, six tuple solutions in Fig. 15c are output in the order b_1 through b_6. In [3], Al-Khalifa et al. also proposed a variant of the StackTree algorithm that outputs tuple solutions in the increasing order of start numbers of ancestor nodes (that is, of A-tagged nodes) by temporarily delaying the output of some tuple solutions that have been found. One such variant is essential for processing twig queries. Consider, for instance, query "$A//B//C$." If we select a query plan $A \bowtie (B \bowtie C)$, then the results of $B \bowtie C$ have to be sorted by B nodes before performing the binary join between A and B.

Skip: no need to sequentially read the entire inverted list. Chien et al. [20] extended the StackTree algorithm

```
(1) min_start = Min(cursor_A.start, cursor_B.start);
    // suppose cursor_X.start = min_start
(2) Clear stack using min_start, i.e. all A nodes in stack with end number
    smaller than min_start are popped out of stack.
(3) If  X = A  Then    Push the node at cursor_A into stack;
    Else   Output tuple solutions into join results. Specifically,
             Case 1 (For the `A/B' query):
                Output a tuple (top, cursor_B), where top is the A node on the
                top of the current stack and top.level + 1 = cursor_B.level;
             Case 2 (For the `A//B' query):
                Output all (a, cursor_B) tuples, where a is any A node in the
                current stack;
(4) advance cursor_X;
```

Fig. 17. Core of the StackTree algorithm.

```
(1) min_start = Min_i{ cursor_i.start };
    // suppose cursor_X.start = min_start
(2) Clear all stacks using min_start, i.e. all nodes in current stacks
    with end number smaller than min_start are popped out of stacks.
(3) If  Stack_parent(X) is not empty   Then
        Push the node at cursor_X into Stack_X with an associated pointer
        to the node on the top of Stack_parent(X);
        If  X is the leaf node of this path query   Then
            Output all tuple solutions implied by current stacks into
            join results through backtracking pointers between stacks;
(4) advance cursor_X;
```

Fig. 18. Core of the PathStack algorithm.

with a "skip" technique, based mainly on a prebuilt B+-tree index on the $start$ numbers of inverted lists, to avoid sequential reads on all data nodes in inverted lists during joins. This might reduce the disk-read costs significantly, since in practice, many data nodes might not form final tuple solutions with other nodes. During the querying process, *skip-descendant* is used when $cursor_B$ falls behind $cursor_A$. In Fig. 16a, $cursor_B$ skips from b_1 to b_i by probing the B+-tree index on $list_B$ by using $a_1.start$ to get the first B-node whose $start$ is larger than $a_1.start$. Similarly, *skip-ancestor* is used when $cursor_A$ falls behind $cursor_B$. Intuitively, in Fig. 16b, $cursor_A$ should skip from a_1 to a_i, where a_i is the first ancestor of b_1. However, the skip-ancestor technique proposed in [20] can only skip from a_1 to a_k by probing the B+-tree index on $list_A$ by using $a_1.end$ to get the first A-node whose $start$ is larger than $a_1.end$. To further improve the efficiency of skipping, that is, to skip directly from a_1 to a_i, Jiang et al. [63] proposed XR-tree indexes rather than regular B+-tree indexes on inverted lists. An XR-tree is basically a B+-tree with all nodes in an inverted list stored in its leaf (data) pages, whereas its internal (index) pages are associated with special *stab lists*, recording the ancestor nodes of the corresponding nodes in its leaf (data) pages. With such stab lists, the first ancestor node a_i of b_1 can be efficiently retrieved by probing the XR-tree using $b_1.start$. Another variant of B+-tree XB-tree [9] is also able to search ancestors efficiently. In an XB-tree, each key K_i in an internal (index) page P is a preassigned interval $(K_i.start, K_i.end)$ that contains all preassigned intervals in $K_i.page$, a child page of P. Experimental work by Li et al. [73] reported that for highly recursive XML data, XB-trees have smaller index size and update cost, as well as better ancestor-searching performance, than XR-trees.

Note that INLJs (Section 4.1.1) can be viewed as a skip technique that skips descendants only (or ancestors only [86]), whereas the skip technique outlined here interleaves ancestor-skip and descendant-skip in a flexible manner throughout the querying process based on the position of $cursor_A$ and $cursor_B$ and, thus, generally has better performance.

4.1.3 The Holistic Approach

StackTree and MPMGJN have to decompose twig queries into multiple binary joins, which might generate a large volume of intermediate query results. For example, for a query plan $(A \bowtie B) \bowtie C$, the result of $A \bowtie B$ has to be written to disk if its size is too large to fit in memory and then has to be read back to memory to join with C after $A \bowtie B$ has been computed. This may result in high disk I/O costs. Motivated by this observation, Bruno et al. [9] proposed a Holistic approach, whose key idea is pipelining, that is, joining multiple inverted lists at a time to avoid generating intermediate join results.

The PathStack approach. The Holistic approach to answering (linear) path queries is a PathStack algorithm [9], whose core is shown in Fig. 18. The framework of the algorithm is somewhat similar to that of StackTree in Fig. 17. The difference is that StackTree uses only one stack to cache nested A nodes. In contrast, PathStack has multiple stacks, one for each node in a path query. In addition, each data node cached in a stack has an associated pointer to a corresponding node in its parent stack in order to track tuple solutions. For an illustration, see Fig. 19.

The TwigStack approach. The Holistic approach to answering general twig queries is a TwigStack algorithm [9], which includes two steps: 1) *deriving path solutions* (that is, the twig query is decomposed into multiple root-to-leaf path queries, and the solutions to these path queries are then derived from the data tree) and 2) *joining path solutions* (that is, the resulting path solutions are then joined to get the final twig solutions).

A simple method for implementing step 1 is to process each path query separately by using PathStack. However, this naive method is generally not efficient, since it might return many redundant path solutions such as (a_1, d_1) (satisfying the path query "$//A//D$") in Fig. 20, which do not contribute to any final twig solutions. To reduce the number of such redundant path solutions, the TwigStack algorithm introduced an additional function $q = getNext()$.

$q = getNext()$ returns a query node q such that q has a subtwig solution, but its parent node does not (we say that a query node q has a *subtwig solution* if given the (query) subtwig rooted at q with all "/" axes replaced by "//" axes, denoted T_q, the cursor (head) nodes of the inverted lists of all query nodes in T_q form a match for T_q). At each step, only $cursor_q$ qualifies for being pushed into the stack. Fig. 20 shows an example. Initially, $getNext()$ advances $cursor_A$ through a_1 to a_2 because $a_1.end < b_1.start$ (thus, a_1 cannot contribute to any final twig solution). Then, at step 1, B has a subtwig solution at b_1, and D has a (trivial) subtwig solution at d_1, but B's and D's parent A does not have a subtwig solution at a_2 because (a_2, b_1, c_1, d_1) does not form a match for T_A. Therefore, we have $getNext() = D$ ($getNext() \neq B$ because $d_1.start < b_1.start$), and thus, d_1 is streamed out of $list_D$. However, d_1 is not pushed into stack D at step 2, since its parent stack A is empty. This way, the redundant path solution (a_1, d_1) is avoided.

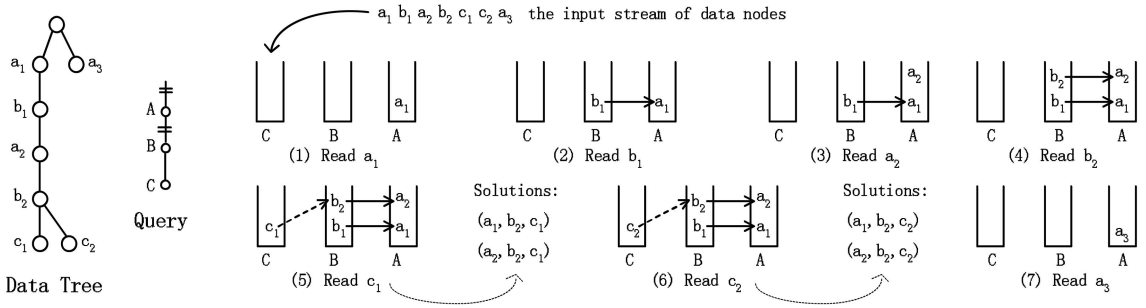


Fig. 19. The PathStack approach: An example.

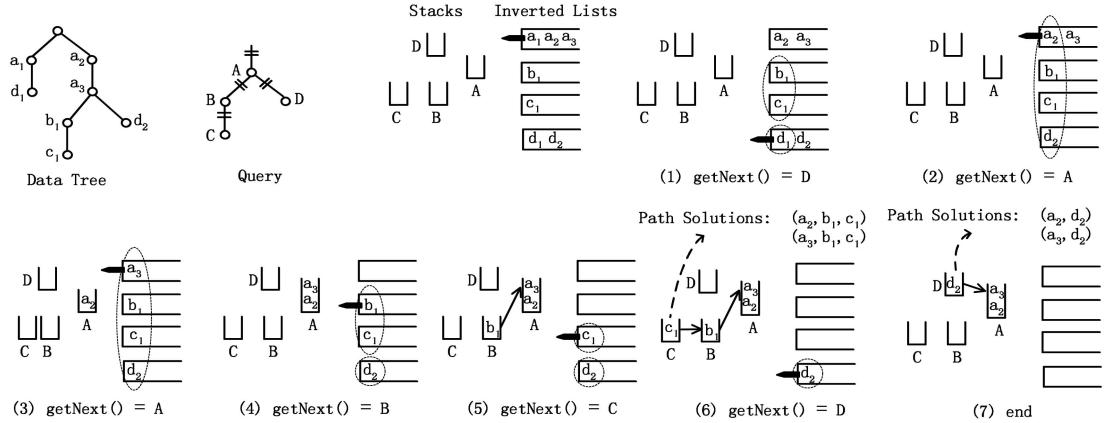


Fig. 20. The TwigStack approach: An example.

Experimental results in [9] show that TwigStack typically has higher query processing performance than StackTree coupled with the optimal join order. Due to these performance advantages, TwigStack has been extended extensively in recent research:

Optimality: no redundant path solutions. In general, TwigStack might still generate redundant path solutions, although it does reduce the number of such solutions compared to the naive method of processing each path query separately by using PathStack. Bruno et al. [9] pointed out that TwigStack is optimal for twig queries with "//" axes only. For instance, in Fig. 20, if we change the "//"-axis between A and D to a "/"-axis, then the (a_2, b_1, c_1) output at step 6 becomes a redundant path solution. However, TwigStack still has to output it as a path solution because at the time of step 6, through checking $cursor_D$ only, TwigStack cannot determine whether a_2 has any D children after $cursor_D = d_2$. Further, Choi et al. [22] showed that any version of TwigStack that *sequentially* reads inverted lists *only once* cannot be optimal for twig queries with arbitrarily mixed "//" and "/" axes. Therefore, research efforts have focused on achieving optimality for some special subclasses of twig queries. Lu et al. [78] proposed a variant of TwigStack, TwigStackList, which "looks ahead" some data nodes in inverted lists and caches them in main memory, to make TwigStack optimal for twig queries where all "/" axes are under nonbranching nodes. More recently, in iTwigJoin [15], Chen et al. extended TwigStack to make it optimal for twig queries with "/" axes only or involving one branching node only by further partitioning each inverted list into multiple sublists based on the level numbers [15] or on root paths [14] of data nodes in the inverted list. We will discuss this method in more detail in Section 4.3.

Skip: no need to sequentially read the entire inverted list. As in the context of binary joins (Section 4.1.2), the skip technique using XB-tree [9] or XR-tree [63] indexes might reduce the disk-read costs of the holistic join significantly. The difference is that in the context of the holistic join, it is important to determine at each step which "broken" query edge should be selected to skip over first. In Fig. 16c, fixing the edge (B, C) first will cause significant disk overhead, since all of b_1 to b_i and c_1 to c_i have to be read sequentially. However, fixing the edge (A, B) first can help avoid such high costs, since $cursor_B$ can skip directly from b_1 to b_i, and then, $cursor_C$ skips directly from c_1 to c_i when fixing the edge (B, C). In [65], Jiang et al. proposed an extension, TSGeneric+, of TwigStack with "skip", which uses histogram and sampling techniques developed in [114] to estimate the average "intermatch distance" (AID) for each pair of query nodes (that is, for each query edge) based on the statistical knowledge on the XML data. The order of fixing broken query edges is in the decreasing order of the AID of query edges. Fontoura et al. [42] further optimize "skip" to reduce the number of expensive *physical* cursor moves (that is, the number of probes of the index prebuilt on inverted lists) by trying to use *virtual* cursor moves as much as possible. For example, in Fig. 16c, it is cheaper to virtually move $cursor_B$ (from b_1) to $cursor_A.start + 1$ ($cursor_A = a_1$) than physically moving it to $b_i.start$, since b_i must be retrieved by probing the index on $list_B$ by using $cursor_A.start$.

Dewey coding: no need to read inverted lists of internal query nodes. In VirtualJ [122], Yang et al. observed that the

inverted lists of internal (nonleaf) query nodes do not need to be accessed during query processing if each node in the inverted lists is expressed using Dewey coding rather than PrePost coding, in the form of $(idPath, tagPath)$. Here, $idPath$ and $tagPath$ are the Dewey vector and the root path of the node, respectively. Note that all ancestors of a node can be retrieved from its $idPath$ and $tagPath$. For instance, for the query in Fig. 20, we only need to access the inverted lists of C and D. Specifically, only those C nodes and D nodes whose $tagPath$ matches the path patterns "$//A//B//C$" and "$//A//D$," respectively, qualify for the join. Furthermore, following the framework of TSGeneric+, which has a cursor for each query node, VirtualJ defines "virtual cursors" for internal query nodes A and B. Here, "virtual cursor" means that $cursor_A$ and $cursor_B$ are derived from the $idPath$ of C nodes and D nodes rather than from $list_A$ and $list_B$, respectively. In the join process, it is not necessary to physically skip ancestors. For instance, in the example in Fig. 16b, only $list_B$ is accessed. This way, a_1 to a_{i-1} are implicitly skipped throughout. Lu et al. [79] independently developed another holistic-join algorithm TJFast, which is also based on Dewey coding but with $(idPath, tagPath)$ pairs "compressed" into one field to save space. Unlike VirtualJ, TJFast is not coupled with the skip-descendant technique. Although Dewey-based join algorithms typically read fewer inverted lists than PrePost-based join algorithms, we observe that the process of reading the $idPath$ and $tagPath$ of data nodes to derive their ancestors is essentially backward navigation (see Section 2.1), which might involve accessing large numbers of query-irrelevant nodes. Therefore, Dewey-based join might be less efficient than PrePost-based join algorithms such as TSGeneric+ when the XML data tree is deep or when ancestor query nodes are very selective (see Fig. 16a for an example).

In other projects, Jiang et al. [62] extended TwigStack to process twig queries with OR predicates, and Bruno et al. [8] extended TwigStack to multiquery processing.

4.1.4 The Index-Graph-Aided (IGA) Approach

The IGA approach is not an independent approach. Rather, it is coupled with other approaches to facilitate query processing [68] whenever some type of index graph G_I of the original XML data—P-index or T-index (Section 2.2)—is available and is small enough (for example, due to the coarse structure of XML data) to navigate efficiently.

As discussed in Section 2.2, T-indexes directly cover twig queries but usually are of larger size than P-indexes. When only P-indexes are available, the IGA approach uses a two-step process to answer twig queries:

1. **Path selection.** A twig query is decomposed into multiple path queries, and IGA navigates over G_I to derive one set of (answer) index nodes I_k for each path query P_k. A relevant graph-querying algorithm can be found in [110].
2. **Path joining.** The data nodes within all I_k's are joined to get answers to the twig query. Note that the data nodes within an index node are stored on disk in the increasing order of their $start$ numbers, as in inverted lists. Therefore, the native join algorithms that we introduced earlier are applicable.

It is easy to see that this two-step process is similar to the PM approach introduced in Section 3.3 (see Fig. 10a for an illustration of the path-decomposition process). The difference is that IGA facilitates step 1 by using P-indexes rather than using any indexes on the path attribute in the Path table (for example, on the ReversedPath attribute in the ReversedPath table; see Fig. 9b), as PM does, while implementing step 2 by using native joins rather than simply pushing down the path-joining task to the SQL engine, as PM does.

IGA has the following benefits: 1) it is not necessary for all nodes in the inverted lists to take part in the join process, which reduces the disk-read costs significantly if path queries are very selective. For example, in Fig. 10a, only the C nodes that satisfy "$//A/B/C$" rather than all C nodes in $list_C$ participate in the join process. 2) If not all query nodes are output nodes, then IGA can also reduce the number of joins, similar to PM.

P-indexes are not the only option for facilitating path selection. We observe that the B+-tree index on PLabel proposed in BLAS (Section 3.3) is essentially an implicit and typically more efficient approximation of P-indexes. First, PLabeling is an implicit approximation because it clusters data nodes by their reversed paths (all data nodes having common prefixes on their reversed paths are clustered in adjacent positions in the $PLabel$ number space) without physically partitioning them into explicit index nodes. Second, PLabeling is more efficient because it is implemented via a B+-tree index, a classical I/O-efficient index that has been widely incorporated into today's mainstream DBMSs. For a given path query, an SQL range query can retrieve answer nodes very efficiently by probing only a few (usually one or two) disk blocks in a B+-tree while always obtaining precise query results. In contrast, P-indexes are ad hoc indexes in today's DBMSs. Furthermore, many existing space-saving P-indexes, for example, A(k)-index, are not precise for all queries, as discussed in Section 2.2, and typically derive a superset of the answer nodes. However, PLabeling also incurs a limitation: It clusters data nodes on disk based on $(PLabel, start)$ rather than on $start$, as in inverted lists. For example, two data nodes, a_1 and a_2, with the reversed paths "D-C-B" and "D-C-A," respectively, are such that $a_1.PLabel > a_2.PLabel$, but it can be that $a_1.start < a_2.start$. Therefore, before native join algorithms are applied, the data nodes derived by path queries, for example, $a_2 - a_1$ by "$//C/D$," have to be first sorted in the increasing order of their $start$ numbers. This might result in significant costs if path queries are not sufficiently selective.

4.2 The Sequence Approach

In ViST [113], Wang et al. proposed a method, which we call the *sequence approach*, for twig pattern matching. The motivation is to avoid the expensive path-solution-joining step used in the second phase of TwigStack by using the entire query twig rather than paths as the basic unit of answering twig queries. The approach works as follows:

1. **Preprocessing: encoding data and query as sequences.** All nodes in the data tree are encoded in a certain way and are then ordered into a sequence S_D. Similarly, all nodes in the query twig are encoded

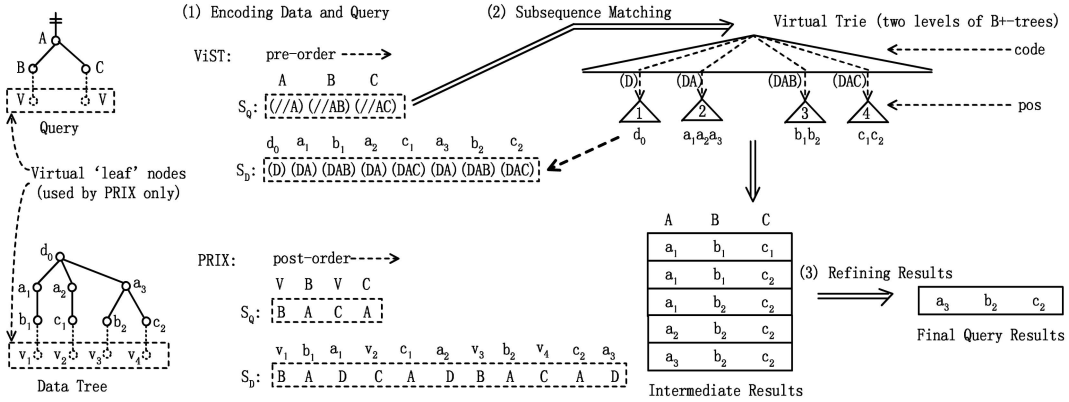


Fig. 21. The Sequence approach: An example.

and ordered into a sequence S_Q. ViST encodes each node by using its root path and orders all nodes by using preorder. PRIX [97] encodes each node by using the tag of its parent node and orders all nodes by using postorder (into Prufer Sequence [95]).

2. **Querying as subsequence matching.** The original task of twig pattern matching is transformed into the task of subsequence matching, that is, into finding in S_D all (noncontiguous) subsequences that match S_Q. This task is implemented via iteratively probing a "virtual trie"[7] index prebuilt on S_D, which is physically implemented as two levels of B+-trees with $(code, pos)$ as keys (see Fig. 21), where $code$ is the code of the node in S_D, and pos indicates the position of the node in S_D. Each index probe $(q.code, d.pos)$, where q is a query node, and d is a data node, first probes the first-level B+-tree by using $q.code$ to reach the corresponding second-level B+-trees and then retrieves the nodes in the second-level B+-trees whose pos is larger than $d.pos$. In Fig. 21, first, an index probe $((//A), 0)$ is issued to reach Tree 2, and all nodes in Tree 2 are retrieved. Then, for a retrieved node, say, a_2, an index probe $((//AB), a_2.pos)$ is issued to reach Tree 3, and b_2 is retrieved. Finally, an index probe $((//AC), b_2.pos)$ is issued to reach Tree 4, and c_2 is retrieved. Thus, (a_2, b_2, c_2) is among the results of subsequence matching.
3. **Postprocessing: refining the results.** Not all subsequence matches returned in step 2 correspond to final twig solutions. For example, in Fig. 21, only (a_3, b_2, c_2) is a twig solution, whereas the other four tuples are not. Therefore, the matches returned in step 2 have to be further refined to get precise query results. ViST uses expensive joins [112], and PRIX uses a sophisticated four-phase process to perform the refinement.

Note that step 2 above might generate many more redundant twig solutions (subsequence matches) than the redundant path solutions generated by TwigStack, which undermines the original goal of the Sequence approach. For example, in Fig. 21, exponentially more redundant subsequence matches would be generated, given that the subtree rooted at a_3 in the data tree is duplicated repeatedly as *following-siblings* of a_3. Wang and Meng [112] addresses this issue by merging steps 2 and 3. The intuition is that at each probing step, only those nodes in the second-level B+-trees that do not violate the "sibling-covering" constraint [112] are retrieved. In the example in Fig. 21, when a_2 is retrieved, and an index probe $((//AB), a_2.pos)$ is issued to reach Tree 3, b_2 is not retrieved because b_2 is sibling-covered by $a_2 : a_3$, a sibling of a_2, is located between a_2 and b_2 in S_D.

The Sequence approach has the following limitations: 1) It supports *ordered* twig queries only. For example, the query in Fig. 21 requires a B-branch to appear before a C-branch. To answer a general (unordered) twig query Q, we have to first transform Q into multiple ordered twig queries with different branch orders and then pose the queries separately against the data tree, which could result in high query costs when Q involves many branches. 2) It may involve a large number of index probes. As we mentioned in Section 4.1.1, index-probe operations might result in high random disk I/O costs. 3) It might repeatedly visit many data nodes unnecessarily. In Fig. 21, b_2 is visited by the index probes from a_1, a_2, and a_3, since $b_2.pos > a_i.pos$. However, the visits via a_1 and a_2 are not necessary and can be avoided in the Join approach. For example, in MPMGJN, the inner loop join of a_1 visits only b_1. The reason is that the Sequence approach uses $(a_i.pos, +\infty)$ to probe indexes, whereas MPMGJN uses $(a_i.start, a_i.end)$ to directly retrieve all descendants of a_i.

4.3 The Navigational Approach

Unlike the Join approach, which searches for twig matches by joining inverted lists, the Navigational approach does that by traversing the data tree along tree edges.

Navigation is the only available option when filtering/querying streaming XML data. In the streaming context, XML data arrive in the form of data streams, without any associated indexes, and the entire XML data tree has to be traversed node by node in the depth-first order, which preserves the original XML document order.

Stream filtering. Recall that in most SDI applications (Section 1.3), a document filter is designed to match each incoming XML document with a collection of subscribed queries. Usually, the subscribed queries are presimulated as

7. For ease of exposition, we consider here only one document sequence. Multiple document sequences form a trie [113].

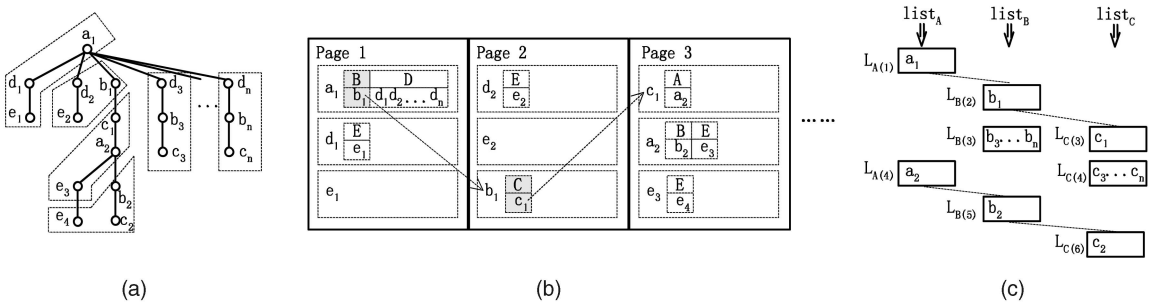


Fig. 22. Navigation versus join: processing query "$//A/B/C$." (a) Data tree. (b) Niagara: navigation over disk storage of data tree. (c) iTwigJoin: prerefined inverted lists.

automata, for example, as nondeterministic finite automata (YFilter [34], [35]), as deterministic finite automata [49], or as deterministic pushdown automata (XPush [55]). Then, the document filter runs the streaming XML document D, which is parsed by a SAX parser [89] sequentially on the fly over (query) automata, as if D were a string, to determine whether D should be accepted by the automata or not. The key idea of improving the performance of such document filtering over a large number of subscribed queries is to share common subexpressions among subscribed queries, for example, common path prefixes (YFilter [34], [35]), common predicates (XPush [55]), and common substrings (XTrie [12]), as much as possible.

Stream querying. A considerable number of stream-querying systems have been developed, with the goal of achieving high CPU performance when evaluating one (rather than multiple, as in SDI applications) XPath/XQuery query over streaming XML data. The stream-querying algorithms have to satisfy stricter requirements than the nonstreaming algorithms reviewed in this survey in that all the data must be read sequentially in one pass, without the help of any indexes. An early XPath stream-querying system XSQ [93], [94] incurs exponential (query) time complexity. Two other XPath stream-querying systems, TurboXPath [66] and TwigM [17], achieve polynomial time complexity. Another XPath stream-querying system XAOS [6] evaluates XPath queries with the parent and ancestor axes (see Section 1.2.1), as well as the commonly used "/" and "//" axes. Recently, some XQuery stream-querying systems have also been developed such as XSM [80], BEA/XQRL [39], FluX [71], Raindrop [107], and FlowGraph [75]. Some of these systems, including XSM, FLuX, and Raindrop, also use the available DTD (Section 3.4) schema information to improve the stream-querying performance and reduce the size of the runtime buffering space.

In the context of querying persistently stored (non-streaming) XML data (which is the focus of this survey), where the goal is to achieve high disk I/O performance, navigation is one of many possible choices. It has been used in many early native XML DBMSs such as Lore [81], [82] and Natix [38]. In most navigation-based systems, the data tree is partitioned into a series of subtrees in the depth-first order (which preserves the original XML document order) rather than into inverted lists based on tag names, and each subtree is stored on a separate disk page (see Fig. 22a). The record of each node on a disk page can include the (tag, id) pairs of all children of that node, which records the information on tree edges for facilitating forward navigation (see Fig. 22b) or can also include the (tag, id) pair of the parent of that node for facilitating backward navigation. However, as discussed in Section 2.1, either forward and backward navigation is generally not very efficient, since each might involve accessing large numbers of query-irrelevant data nodes scattered across a number of disk pages and thus cause high disk I/O cost. This fact motivates the development of the Join approach, which joins only query-relevant inverted lists by using numbering schemes.

Recently, Niagara [56] and NoK [125] have called for a revival of navigation. The authors argue that navigation might be more efficient than join when processing the "/" axis. For example, in Fig. 22, joining $list_A$, $list_B$, and $list_C$ is not efficient, since b_3 through b_n in $list_B$ are proper descendants, rather than children, of a_1. In contrast, navigation can avoid unnecessary accesses to b_3 through b_n (see Fig. 22b). Motivated by this, Niagara answers twig queries in a mixed mode of navigation and join. In particular, it constructs a sophisticated cost model to determine the navigation part and the join part of a query. The experimental results in [56] seem to indicate that navigation is always selected for the "/" axis and join for the "//" axis. For example, for query "$/A/B/D//F$," Niagara selects navigation for "$/A/B/D$" and then join for "$//F$." It is explicitly proposed for NoK [125] to use navigation for all cases of the "/" axis, and join for all cases of the "//" axis in a query.

We beg to differ in that we think that joins could be more efficient than navigation, even when processing the "/" axis, provided that iTwigJoin [15] (a variant of TwigStack) is used. As mentioned in Section 4.1.3, iTwigJoin achieves optimality for twig queries with only the "/" axis by partitioning each inverted list into multiple sublists based on the $level$ values of data nodes. For example, in Fig. 22c, $list_B$ is partitioned into three sublists: $L_{B(2)}$, $L_{B(3)}$, and $L_{B(5)}$. This way, only level-relevant sublists are joined with each other when processing the "/" axes, omitting unnecessary accesses to level-irrelevant sublists, for example, $L_{B(3)}$ and $L_{C(4)}$. It is easy to see that iTwigJoin has better performance than navigation, since it clusters nodes on disk based on their $(tag, level, start)$ triples, whereas the disk pages retrieved in navigation, for example, Page 1 in Fig. 22b, may include many query-irrelevant nodes such as d_1 and e_1.

At the same time, recall from Section 4.1.3 that some competitive Join techniques such as VirtualJ, incorporate

backward navigation implicitly in their join process. Navigation appears there in the form of exploring the Dewey vectors of data nodes stored in inverted lists to reduce the number of joins and to achieve good query performance when the XML data tree is shallow and when the descendant query nodes are very selective.

We also note that when processing a sequence of consecutive "/" axes in a query, say, "$//A/B/C$," navigation is not really necessary if some type of path index such as PLabeling (Section 3.3) is available. In such case, an SQL range query on PLabel could be much more efficient than navigation.

4.4 Summary

The Join approach provides an efficient native implementation of θ-joins used in the relational approach. As discussed in Sections 4.2 and 4.3, in general, Join has performance advantages over Sequence Matching and Navigation. Some experimental work by Moro et al. [86] has also partially verified this point.

Specifically, among the Join techniques: 1) the performance of Holistic is better than that of MPMGJN or of StackTree; that is, the pipelining-join strategy tends to dominate the binary-join strategy, 2) when an index graph or a PLabeling index is available, it could be used to reduce the number of joins or to "shorten" the inverted lists before the joins, and 3) generally, TSGeneric+ and VirtualJ seem to be the most competitive Holistic techniques. TSGeneric+, a variant of TwigStack with "skip," reduces the length of the inverted lists accessed during the joins. VirtualJ, a Dewey variant of TSGeneric+, reduces both the number and length of the accessed inverted lists. However, when the XML data tree is deep or when the ancestor query nodes are very selective, VirtualJ may yield to TSGeneric+ in performance.

5 Conclusion

We have reviewed several major techniques for XML twig-query processing and categorized them into two classes, that is, the relational approach and the native approach:

1. In the relational approach, XML data are loaded into relational databases, and XML twig queries are transformed into SQL queries over relational data. All query processing are typically pushed into existing relational query optimizers, rendering major extra implementation unnecessary. However, current RDBMSs do not support θ-joins efficiently, although θ-joins have become an important component of answering XML queries efficiently. Among the relational approaches, the BLAS approach—an extension of the basic RP approach with PLabeling—seems to be the best with respect to query processing performance.
2. In the native approach, XML data are stored on disk in the form of inverted lists, sequences, or trees, and native algorithms are developed to further improve XML twig-query performance. Among the native approaches, the Join approach provides an efficient native implementation of θ-joins used in the relational approach and seems to show the most promising query performance in practice. However, in the native approach, many important RDBMS components such as storage management, access methods, query processing and optimization, and concurrency control and recovery have to be built from scratch.

A good trade-off between the relational approach and the native approach would be storing XML data in the form of inverted lists by using existing relational databases, coupled with integrating efficient native join algorithms for XML twig queries into existing relational query optimizers. That is,

$$\begin{aligned}&''\text{relational storage}'' \text{ of XML data}\\ &+ ''\text{native processing}'' \text{ of XML queries.}\end{aligned}$$

As a result, by using extended relational query optimizers, RDBMSs would be able to process XML twig queries more efficiently, whereas other important existing components of RDBMSs could be fully reused. One such integration could result in higher XML query processing performance and could also significantly reduce system-reengineering costs. This framework happens to be the choice of current mainstream commercial RDBMSs, including IBM DB2 [7], [59], Microsoft SQL Server [88], [90], and Oracle DB [87], on their way toward efficient XML data querying and management.

Acknowledgments

The authors are grateful to Jun Yang and to the anonymous reviewers for their valuable comments and suggestions on earlier drafts of this survey. They also would like to thank the US National Science Foundation (NSF) for supporting this work under NSF Faculty Early Career Development (CAREER) Award 0447742 and NSF IIS Grant 0307072.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[2] R. Agrawal, A. Borgida, and H.V. Jagadish, "Management of Transitive Relationships in Large Data and Knowledge Bases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '89),* 1989.

[3] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. 18th IEEE Int'l Conf. Data Eng. (ICDE '02),* 2002.

[4] M. Altinel and M.J. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," *Proc. 26th Int'l Conf. Very Large Data Bases (VLDB '00),* 2000.

[5] S. Amer-Yahia, L.V.S. Lakshmanan, and S. Pandit, "FleXPath: Flexible Structure and Full-Text Querying for XML," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, "Streaming XPath Processing with Forward and Backward Axes," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.

[7] K.S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G.M. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T.C. Truong, B. Van der Linden, B. Vickery, and C. Zhang, "System RX: One Part Relational, One Part XML," *Proc. 24th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05),* 2005.

[8] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava, "Navigation- vs. Index-Based XML Multi-Query Processing," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.

[9] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proc. 21st ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* 2002.

[10] Online Computer Library Center, Dewey Decimal Classification, http://www.oclc.org/dewey/, 2006.

[11] D.D. Chamberlin, "XQuery: An XML Query Language," *IBM Systems J.,* vol. 41, no. 4, 2002.

[12] C.Y. Chan, P. Felber, M.N. Garofalakis, and R. Rastogi, "Efficient Filtering of XML Documents with XPath Expressions," *Proc. 18th IEEE Int'l Conf. Data Eng. (ICDE '02),* 2002.

[13] Q. Chen, A. Lim, and K.W. Ong, "D(K)-Index: An Adaptive Structural Summary for Graph-Structured Data," *Proc. 22nd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[14] T. Chen, T.W. Ling, and C.Y. Chang, "Prefix Path Streaming: A New Clustering Method for Optimal Holistic XML Twig Pattern Matching," *Proc. 15th Int'l Conf. Database and Expert Systems Applications (DEXA '04),* 2004.

[15] T. Chen, J. Lu, and T.W. Ling, "On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques," *Proc. 24th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05),* 2005.

[16] Y. Chen, S.B. Davidson, and Y. Zheng, "BLAS: An Efficient XPath Processing System," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[17] Y. Chen, S.B. Davidson, and Y. Zheng, "An Efficient XPath Query Processor for XML Streams," *Proc. 22nd IEEE Int'l Conf. Data Eng. (ICDE '06),* 2006.

[18] Z. Chen, H.V. Jagadish, L.V.S. Lakshmanan, and S. Paparizos, "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[19] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava, "Index Structures for Matching XML Twigs Using Relational Query Processors," *Proc. Second Int'l Workshop XML Schema and Data Management (XSDM '05),* 2005.

[20] S.-Y. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, and C. Zaniolo, "Efficient Structural Joins on Indexed XML Documents," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* 2002.

[21] B. Choi, "What Are Real DTDs Like," *Proc. Fifth Int'l Workshop Web and Databases (WebDB '02),* 2002.

[22] B. Choi, M. Mahoui, and D. Wood, "On the Optimality of Holistic Algorithms for Twig Queries," *Proc. 14th Int'l Workshop Database and Expert Systems Applications (DEXA '03),* 2003.

[23] C.-W. Chung, J.-K. Min, and K. Shim, "APEX: An Adaptive Path Index for XML Data," *Proc. 21st ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* 2002.

[24] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and Distance Queries via 2-Hop Labels," *SIAM J. Computing,* vol. 32, pp. 1338-1355, 2003.

[25] W3C Consortium, http://www.w3.org, 2006.

[26] W3C Consortium, *Guide to the W3C XML Specification (XMLspec) DTD, Version 2.1,* http://www.w3.org/XML/1998/06/xmlspec-report.htm, 2006.

[27] W3C Consortium, XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/, 2006.

[28] W3C Consortium, XML Query Use Cases, http://www.w3.org/TR/xquery-use-cases/, 2006.

[29] W3C Consortium, XML Schema, http://www.w3.org/XML/Schema, 2006.

[30] W3C Consortium, XQuery 1.0: An XML Query Language, http://www.w3.org/TR/xquery/, 2006.

[31] B. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, and M. Shadmon, "A Fast Index for Semistructured Data," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01),* 2001.

[32] D. DeHaan, D. Toman, M.P. Consens, and M.T. Ozsu, "A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding," *Proc. 22nd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[33] A. Deutsch, Y. Papakonstantinou, and Y. Xu, "The NEXT Logical Framework for XQuery," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[34] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P.M. Fischer, "Path Sharing and Predicate Evaluation for High-Performance XML Filtering," *ACM Trans. Database Systems,* vol. 28, pp. 467-516, 2003.

[35] Y. Diao and M.J. Franklin, "Query Processing for High-Volume XML Message Brokering," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[36] P.F. Dietz, "Maintaining Order in a Linked List," *Proc. 14th ACM Symp. Theory of Computing,* 1982.

[37] M. Fernandez et al., "Galax: An Implementation of XQuery," http://www.galaxquery.org/, 2006.

[38] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann, "Anatomy of a Native XML Base Management System," *VLDB J.,* vol. 11, no. 4, pp. 292-314, 2002.

[39] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M.J. Carey, A. Sundararajan, and G. Agrawal, "The BEA/XQRL Streaming XQuery Processor," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[40] D. Florescu and D. Kossmann, "A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database," Technical Report 3684, INRIA, 1999.

[41] D. Florescu and D. Kossmann, "Storing and Querying XML Data Using an RDMBS," *IEEE Data Eng. Bull.,* vol. 22, pp. 27-34, 1999.

[42] M. Fontoura, V. Josifovski, E.J. Shekita, and B. Yang, "Optimizing Cursor Movement in Holistic Twig Joins," *Proc. 14th Int'l Conf. Information and Knowledge Management (CIKM '05),* 2005.

[43] N. Fuhr and K. GroBjohann, "XIRQL: A Query Language for Information Retrieval in XML Documents," *Proc. 24th ACM Int'l Conf. Research and Development in Information Retrieval (SIGIR '01),* 2001.

[44] R. Goldman and J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases," *Proc. 23rd Int'l Conf. Very Large Data Bases (VLDB '97),* 1997.

[45] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* 2002.

[46] G. Gottlob, C. Koch, and R. Pichler, "The Complexity of XPath Query Evaluation," *Proc. 22nd ACM Symp. Principles of Database Systems (PODS '03),* 2003.

[47] G. Gottlob, C. Koch, and R. Pichler, "XPath Query Evaluation: Improving Time and Space Efficiency," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.

[48] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," *ACM Trans. Database Systems,* vol. 30, no. 2, pp. 444-491, 2005.

[49] T.J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML Streams with Deterministic Automata and Stream Indexes," *ACM Trans. Database Systems,* vol. 29, pp. 752-788, 2004.

[50] T. Grust, "Accelerating XPath Location Steps," *Proc. 21st ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* 2002.

[51] T. Grust, S. Sakr, and J. Teubner, "XQuery on SQL Hosts," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[52] T. Grust, M. van Keulen, and J. Teubner, "Staircase Join: Teach a Relational DBMS to Watch Its (Axis) Steps," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[53] T. Grust, M. van Keulen, and J. Teubner, "Accelerating XPath Evaluation in Any RDBMS," *ACM Trans. Database Systems,* vol. 29, pp. 91-131, 2004.

[54] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "XRANK: Ranked Keyword Search over XML Documents," *Proc. 29th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[55] A.K. Gupta and D. Suciu, "Stream Processing of XPath Queries with Predicates," *Proc. 29th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[56] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A.N. Rao, F. Tian, S. Viglas, Y. Wang, J.F. Naughton, and D.J. DeWitt, "Mixed Mode XML Query Processing," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[57] H. He, H. Wang, J. Yang, and P.S. Yu, "Compact Reachability Labeling for Graph-Structured Data," *Proc. 14th Int'l Conf. Information and Knowledge Management (CIKM '05),* 2005.

[58] H. He and J. Yang, "Multiresolution Indexing of XML for Frequent Queries," *Proc. 20th IEEE Int'l Conf. Data Eng. (ICDE '04),* 2004.

[59] IBM, http://www-306.ibm.com/software/data/db2/9/, 2006.

[60] H.V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Paparizos, J.M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A Native XML Database," *VLDB J.,* vol. 11, pp. 274-291, 2002.

[61] H.V. Jagadish, L.V.S. Lakshmanan, D. Srivastava, and K. Thompson, "TAX: A Tree Algebra for XML," *Proc. Eighth Int'l Workshop Databases and Programming Languages (DBPL '01),* 2001.

[62] H. Jiang, H. Lu, and W. Wang, "Efficient Processing of Twig Queries with OR-Predicates," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[63] H. Jiang, H. Lu, W. Wang, and B.C. Ooi, "XR-Tree: Indexing XML Data for Efficient Structural Joins," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.

[64] H. Jiang, H. Lu, W. Wang, and J.X. Yu, "Path Materialization Revisited: An Efficient Storage Model for XML Data," *Proc. 13th Australasian Database Conf. (ADC '02) ,* 2002.

[65] H. Jiang, W. Wang, H. Lu, and J.X. Yu, "Holistic Twig Joins on Indexed XML Documents," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[66] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML Streams," *VLDB J.,* vol. 14, no. 2, pp. 197-210, 2005.

[67] R. Kaushik, P. Bohannon, J.F. Naughton, and H.F. Korth, "Covering Indexes for Branching Path Queries," *Proc. 21st ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* 2002.

[68] R. Kaushik, R. Krishnamurthy, J.F. Naughton, and R. Ramakrishnan, "On the Integration of Structure Indexes and Inverted Lists," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[69] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data," *Proc. 18th IEEE Int'l Conf. Data Eng. (ICDE '02),* 2002.

[70] M.H. Kay, "SAXON: The XSLT and XQuery Processor," http://saxon.sourceforge.net/, 2006.

[71] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier, "Schema-Based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[72] R. Krishnamurthy, R. Kaushik, and J.F. Naughton, "XML-SQL Query Translation Literature: The State of the Art and Open Problems," *Proc. First Int'l XML Database Symp. (XSym '03),* 2003.

[73] H. Li, M.-L. Lee, W. Hsu, and C. Chen, "An Evaluation of XML Indexes for Structural Join," *SIGMOD Record,* vol. 33, no. 3, pp. 28-33, 2004.

[74] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01),* 2001.

[75] X. Li and G. Agrawal, "Efficient Evaluation of XQuery over Streaming Data," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.

[76] Y. Li, C. Yu, and H.V. Jagadish, "Schema-Free XQuerys," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[77] S. Liu, Q. Zou, and W.W. Chu, "Configurable Indexing and Ranking for XML Information Retrieval," *Proc. 27th Int'l ACM Conf. Research and Development in Information Retrieval (SIGIR '04),* 2004.

[78] J. Lu, T. Chen, and T.W. Ling, "Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-Ahead Approach," *Proc. 13th Int'l Conf. Information and Knowledge Management (CIKM '04),* 2004.

[79] J. Lu, T.W. Ling, C.Y. Chan, and T. Chen, "From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.

[80] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou, "A Transducer-Based XML Query Processor," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB '02),* 2002.

[81] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A Database Management System for Semistructured Data," *SIGMOD Record,* vol. 26, no. 3, pp. 54-66, 1997.

[82] J. McHugh and J. Widom, "Query Optimization for XML," *Proc. 25th Int'l Conf. Very Large Data Bases (VLDB '99),* 1999.

[83] R. Milner, "A Calculus for Communicating Processes," *Lecture Notes in Computer Science 92,* Springer-Verlag, 1980.

[84] T. Milo and D. Suciu, "Index Structures for Path Expressions," *Proc. Seventh Int'l Conf. Database Theory (ICDT '99),* 1999.

[85] J.-K. Min, M.-J. Park, and C.-W. Chung, "XPRESS: A Queriable Compression for XML Data," *Proc. 22nd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[86] M.M. Moro, Z. Vagena, and V.J. Tsotras, "Tree-Pattern Queries on a Lightweight XML Processor," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.

[87] R. Murthy, Z.H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy, "Towards an Enterprise XML Architecture," *Proc. 24th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05),* 2005.

[88] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury, "ORDPATHs: Insert-Friendly XML Node Labels," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[89] SAX Project Organization, SAX: Simple API for XML, http://www.saxproject.org/, 2004.

[90] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. Zolotov, "Indexing XML Data Stored in a Relational Database," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[91] S. Paparizos, Y. Wu, L.V.S. Lakshmanan, and H.V. Jagadish, "Tree Logical Classes for Efficient Evaluation of XQuery," *Proc. 23rd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '04),* 2004.

[92] D. Park, "Concurrency and Automata on Infinite Sequences," *Proc. Fifth GI Conf. Theoretical Computer Science,* pp. 167-183, 1981.

[93] F. Peng and S.S. Chawathe, "XPath Queries on Streaming Data," *Proc. 22nd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[94] F. Peng and S.S. Chawathe, "XSQ: A Streaming XPath Engine," *ACM Trans. Database Systems,* vol. 30, pp. 577-623, 2005.

[95] H. Prufer, "Neuer Beweis Eines Satzes Uber Permutationen," *Archiv fur Mathematik und Physik,* vol. 27, pp. 142-144, 1918.

[96] P. Ramanan, "Covering Indexes for XML Queries: Bisimulation-Simulation = Negation," *Proc. 29th Int'l Conf. Very Large Data Bases (VLDB '03),* 2003.

[97] P. Rao and B. Moon, "PRIX: Indexing and Querying XML Using Prufer Sequences," *Proc. 20th IEEE Int'l Conf. Data Eng. (ICDE '04),* 2004.

[98] F. Rizzolo and A.O. Mendelzon, "Indexing XML Data with ToXin," *Proc. Fourth Int'l Workshop Web and Databases (WebDB '01),* 2001.

[99] G. Salton and M.J. McGill, *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[100] R. Schenkel, A. Theobald, and G. Weikum, "HOPI: An Efficient Connection Index for Complex XML Document Collections," *Proc. Ninth Int'l Conf. Extending Database Technology (EDBT '04),* 2004.

[101] R. Schenkel, A. Theobald, and G. Weikum, "Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections," *Proc. 21st IEEE Int'l Conf. Data Eng. (ICDE '05),* 2005.

[102] A. Schmidt, M.L. Kersten, M. Windhouwer, and F. Waas, "Efficient Relational Storage and Retrieval of XML Documents," *Proc. Third Int'l Workshop Web and Databases (WebDB '00),* 2000.

[103] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '79),* 1979.

[104] J. Shanmugasundaram, E.J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J.F. Naughton, and I. Tatarinov, "A General Technique for Querying XML Documents Using a Relational Database System," *SIGMOD Record,* vol. 30, pp. 20-26, 2001.

[105] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, and J.F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities," *Proc. 25th Int'l Conf. Very Large Data Bases (VLDB '99),* 1999.

[106] A. Silberstein, H. He, K. Yi, and J. Yang, "BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data," *Proc. 21st IEEE Int'l Conf. Data Eng. (ICDE '05),* 2005.

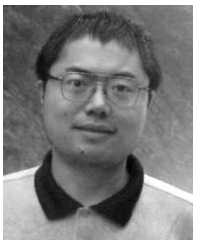
[107] H. Su, E.A. Rundensteiner, and M. Mani, "Semantic Query Optimization for XQuery over XML Streams," *Proc. 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.

[108] I. Tatarinov, S. Viglas, K.S. Beyer, J. Shanmugasundaram, E.J. Shekita, and C. Zhang, "Storing and Querying Ordered XML Using a Relational Database System," *Proc. 21st ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02),* 2002.

[109] F. Tian, D.J. DeWitt, J. Chen, and C. Zhang, "The Design and Performance Evaluation of Alternative XML Storage Strategies," *SIGMOD Record,* vol. 31, no. 1, pp. 5-10, 2002.

[110] Z. Vagena, M.M. Moro, and V.J. Tsotras, "Twig Query Processing over Graph-Structured XML Data," *Proc. Seventh Int'l Workshop Web and Databases (WebDB '04),* 2004.

[111] H. Wang, H. He, J. Yang, P.S. Yu, and J.X. Yu, "Dual Labeling: Answering Graph Reachability Queries in Constant Time," *Proc. 22nd IEEE Int'l Conf. Data Eng. (ICDE '06),* 2006.
[112] H. Wang and X. Meng, "On the Sequencing of Tree Structures for XML Indexing," *Proc. 21st IEEE Int'l Conf. Data Eng. (ICDE '05),* 2005.
[113] H. Wang, S. Park, W. Fan, and P.S. Yu, "ViST: A Dynamic Index Method for Querying XML Data by Tree Structures," *Proc. 29th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.
[114] W. Wang, H. Jiang, H. Lu, and J.X. Yu, "Containment Join Size Estimation: Models and Methods," *Proc. 22nd ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.
[115] W. Wang, H. Jiang, H. Lu, and J.X. Yu, "PBiTree Coding and Efficient Processing of Containment Joins," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.
[116] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li, "Efficient Processing of XML Path Queries Using the Disk-Based F&B Index," *Proc 31st Int'l Conf. Very Large Data Bases (VLDB '05),* 2005.
[117] F. Weigel, H. Meuss, F. Bry, and K.U. Schulz, "Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data," *Proc. 26th European Conf. IR Research (ECIR '04),* 2004.
[118] F. Weigel, K.U. Schulz, and H. Meuss, "The BIRD Numbering Scheme for XML and Tree Databases—Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations," *Proc. Third Int'l XML Database Symp. (XSym '05),* 2005.
[119] Y. Wu, J.M. Patel, and H.V. Jagadish, "Estimating Answer Sizes for XML Queries," *Proc. Eighth Int'l Conf. Extending Database Technology (EDBT '02),* 2002.
[120] Y. Wu, J.M. Patel, and H.V. Jagadish, "Structural Join Order Selection for XML Query Optimization," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.
[121] Y. Xu and Y. Papakonstantinou, "Efficient Keyword Search for Smallest LCAs in XML Databases," *Proc. 24th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '05),* 2005.
[122] B. Yang, M. Fontoura, E.J. Shekita, S. Rajagopalan, and K.S. Beyer, "Virtual Cursors for XML Joins," *Proc. 13th Int'l Conf. Information and Knowledge Management (CIKM '04),* 2004.
[123] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura, "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases," *ACM Trans. Internet Technology,* vol. 1, pp. 110-141, 2001.
[124] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.M. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proc. 20th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '01),* 2001.
[125] N. Zhang, V. Kacholia, and M.T. Ozsu, "A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML," *Proc. 20th IEEE Int'l Conf. Data Eng. (ICDE '04),* 2004.



Gang Gou received the BSc degree in computer science from Nankai University, China, and the MPhil degree in systems engineering from the Chinese University of Hong Kong in 2000 and 2003, respectively. He is a fourth-year PhD student in the Department of Computer Science at North Carolina State University. He is currently affiliated with the Database Research Group, North Carolina State University, and is advised by Dr. Rada Chirkova. His primary research interest is database systems, with specialization in designing high-performance algorithms for efficiently managing and querying large-scale data, which ranges from structured data stored in relational databases to semistructured XML data. He has published papers on leading database conferences and journals, including the ACM SIGMOD International Conference on Management of Data and the *Transactions on Knowledge and Data Engineering*. He was elected to the Phi Kappa Phi Honor Society in 2005.

Rada Chirkova received the BSc and MSc degrees in applied mathematics from Moscow State University in 1991 and 1995, respectively, and the PhD degree in computer science from Stanford University in 2002. She is an assistant professor in the Computer Science Department at North Carolina State University. Her research interests are databases. Some of her recent or current projects include query rewriting (including query optimization) and view selection for a variety of query, view, and rewriting languages on relational and XML data, using restructured views (such as tables resulting from the "pivot" operation in relational database management system (DBMS)) in efficient query processing, and query processing for data integration and interoperability. She has served on the program committees of leading database conferences, including the ACM SIGMOD International Conference on Management of Data, the International Conference on Very Large Data Bases (VLDB), and the ACM Symposium on Principles of Database Systems (PODS). She is a recipient of the US National Science Foundation Faculty Early Career Development (CAREER) Award.