XML Query Processing: A Survey *

Gang Gou, Rada Chirkova

Department of Computer Science
North Carolina State University
Raleigh, NC USA 27695-8207

Email: ggou@ncsu.edu, chirkova@csc.ncsu.edu

May 10, 2005

Abstract

XML (Extensible Markup Language) is emerging as a de facto standard for information exchange among various applications on the web because of its inherent data self-describing capability and flexibility of organizing data. With increased impact of XML on information exchange, it is particularly important to develop high-performance techniques to query large XML data repositories efficiently.

The core of XML query processing is twig pattern matching, i.e. finding from XML documents all matches that satisfy the twig (or path) pattern specified by a given query. In this survey we will review and compare major techniques for processing XML twig queries. We categorize these techniques into three classes based on the storage format of XML data. First, we review the *file approach*, in which XML data have to be stored in commonly used *flat files*, in the form of just original XML documents, for special-purpose applications. Then, we review the *relational approach*, in which XML data are stored in *relational databases* so that all existing important techniques that have been developed for relational databases can be fully reused and so no extra development efforts are needed. Finally, we review the *native approach*, in which XML data are stored in *inverted lists* and native algorithms are developed to further improve XML query processing performance.

To the best of our knowledge, this is the first survey work that systematically reviews, classifies, and compares state-of-the-art techniques for XML query processing.

*All copyrights of this technical report are reserved by the authors and North Carolina State University.

1 Introduction

XML (Extensible Markup Language) is emerging as a *de facto* standard for information exchange among various applications on the web because of its inherent data self-describing capability and flexibility of organizing data [Gro04a].

First, data in XML documents are self-describing. Similar to the familiar HTML (HyperText Markup Language), XML is based on *nested* tags. Figure 1 (a) shows an example of an XML document, which records information about publishers. However, unlike HTML, in which tags associated with data are used to express the *presentation style* (e.g. font styles) of data, tags in XML are used to describe the *semantics* of data. For example, Line 3 in Figure 1 (a) says that '*Cambridge*' is an address of a publisher named '*MITPress*'. Therefore, when an application receives an XML document from another application over the web, it can understand the content of this XML document, since data in XML documents are self-describing.

Second, XML is flexible in organizing data. The nested hierarchy of tags structurizes the content of XML documents. The role of nested tags is somewhat similar to *schemas* in relational databases. However, the nested XML model is more flexible than the flat relational model. The same objects in an XML document might have different kinds of sub-objects or different number of sub-objects of the same kind. For example, in Figure 1 (a), the first *publisher* has an *address* sub-element but the second *publisher* does not. The *book* under the first *publisher* has two *author* sub-elements but the *book* under the second *publisher* has only one *author* sub-element.

```
<Publishers>
  <Publisher @name=' MIT Press' >
     <address> Cambridge </address>
     <book>
         <title> database </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) XML document without ID/IDREF

```
<Publishers>
  <Publisher @name=' MIT Press' >
     <address> Cambridge </address>
     <book>
         <title> database </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) XML document with ID/IDREF

Figure 1: XML documents

1.1 Data Model

1.1.1 Basic Model: Tree

The basic data model of XML is a directed, rooted, labeled, and ordered tree. Figure 2 (a) and (b) shows the XML data tree of the XML document in Figure 1 (a) [1]. Figure 2 (a) is based on a *node-labeled* model where labels are on nodes, and Figure 2 (b) is based on an *edge-labeled* model where labels are on edges. These two models are equivalent. Most research papers use the node-labeled model, while the edge-labeled model is also used in some scenarios, such as in the *Edge* approach that will be introduced in Section 4.2. Here we explain the XML data tree based on the node-labeled model, and analogous explanations can also be applied to the edge-labeled model.

There are three classes of nodes in a data tree. (1) *Element* Node (internal node). This class of nodes correspond to tags in XML documents, such as *publisher*, *address*, etc. Labels on element nodes are just tags in XML documents. (2) *Attribute* Node (internal node). This class of nodes correspond to attributes in XML documents, such as '*@name*' under the first *publisher* element. In contrast to element nodes, attribute nodes are not nested (i.e. an attribute cannot have any sub-elements), are not repeatable (i.e. two same-name attributes cannot occur under one element), and are unordered (i.e. attributes of an element can freely interchange their occurrence locations under this element). (3) *Value* Node (leaf node). This class of nodes correspond to data values in XML documents such as '$MITPress'$', '*database*'', etc.

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

1.1.2 Extended Model: DAG and General Graph

XML documents allow users to define ID/IDREF attributes of elements, where the *id* attribute is used to uniquely identify an element and *idref* attributes are used to refer to other elements which are explicitly identified by their id attributes. ID/IDREF attributes increase the flexibility of the XML model so that elements in XML documents may directly refer to each other freely. Figure 1 (b) shows an XML document with ID/IDREF attributes, where the newly introduced ID/IDREF attributes are underlined.

Therefore, in addition to original *tree edges* in XML data trees which describe main *skeleton* structural relationships in XML documents, *ID/IDREF edges* are also introduced into the XML data model to represent *direct reference* relationships between elements, which extends the original tree model to DAG (Directed Acyclic Graph) or even more general graph with cycles. Figure 2 (c) is just a graph with cycles, which corresponds to the XML document in Figure 1 (b).

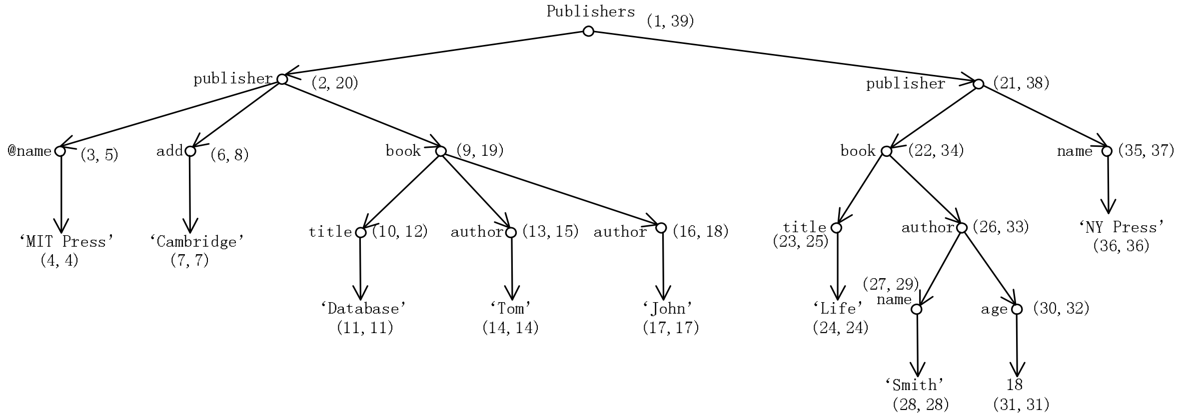
1.2 XML Queries

Unlike *keyword search* in *text retrieval*, which concerns only *contents* of text documents, XML queries concern *structure* as well as *contents* of XML documents.

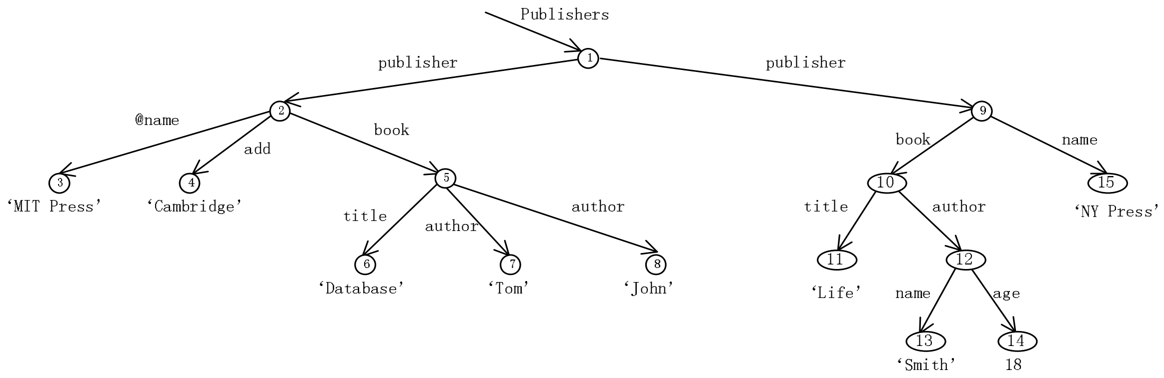
1.2.1 XPath

XPath [Gro04c] is a basic XML query language that is used to select *nodes* from XML documents such that the path from the root to each selected node satisfies a specified *pattern*. A simple XPath query is

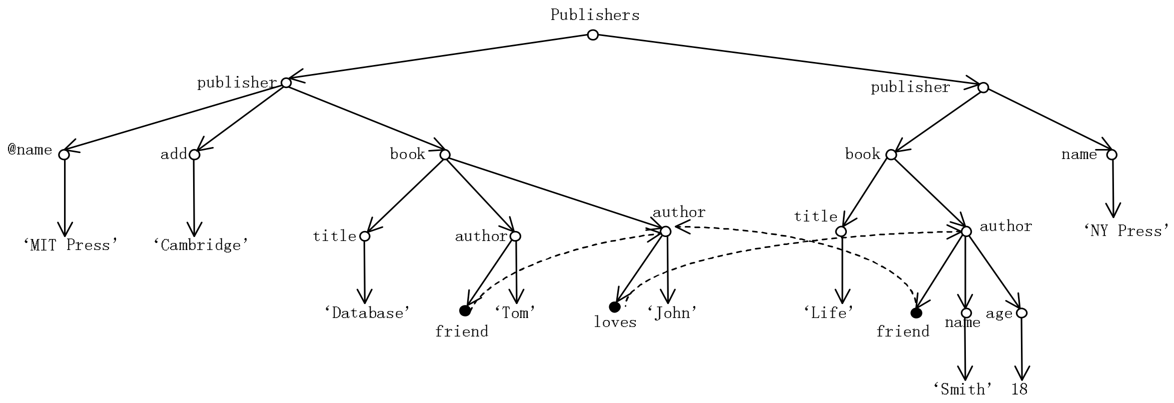
[1] We defer discussing the pairs of numbers adorning nodes until later.



(a) Node-labeled XML Data Tree



(b) Edge-labeled XML Data Tree



(c) Node-labeled XML Data Graph (with ID/IDREF edges)

Figure 2: XML data model

specified by a sequence of alternate axes and tags. Two commonly used axes are *child axis* '/' where 'A/B' denotes selecting B-tagged child nodes of A-tagged nodes, and *descendant axis* '//' where '$A//B$' denotes selecting B-tagged descendant nodes of A-tagged nodes. An example XPath query is "$/Publisher//title$" (its standard form should be "$root/Publisher//title$", but "root" is always omitted for simplicity), which returns all book titles of all publishers. The result of this query against the data tree in Figure 2 (a) is a set of *title* nodes that have values 'Database' and 'Life'.

The query pattern specified by the XPath query above is a simple *path pattern* shown in Figure 3 (a) where the arrow with '=' denotes the '//' axis. Generally, an XPath query can specify a more complex *tree pattern* (also called *twig pattern*) by introducing *selection predicates* into XPath expressions. One such example is "$/Publisher[@name = \text{'}MITPress\text{'}]/book/title$", in which '$/Publisher/book/title$' is the *main path* of this query and the content between '[' and ']' is a *selection predicate*. This query returns all book titles of the publisher named 'MIT Press'. The pattern of this query is shown in Figure 3 (b). Generally, multiple selection predicates might be involved in XPath queries.

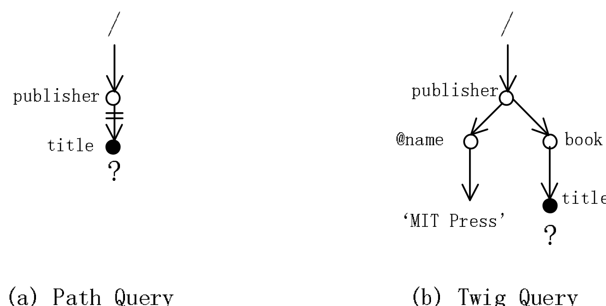


Figure 3: Query pattern

1.2.2 XQuery

XQuery [Gro04e, Cha02] is another popular XML query language, which is an extension to XPath and is more powerful than XPath. It is a functional language comprised of FLWR (For-Let-Where-Return) clauses that can be nested and composed with full generality. *For* and *Let* clauses bind nodes selected by XPath expressions to user-defined *node variables*. *Where* clauses specify selection or join predicates on *node variables*. *Return* clauses operate on *node variables* to construct a new XML document as the query result. Figure 4 (a) shows a simple XQuery, which groups books by their publisher addresses. The query pattern is shown in Figure 4 (b), and the format of the resulting XML document is shown in Figure 4 (c) where the '$*$' edge means that a *books* node might have multiple *book* nodes as children. From this example we can find that XQuery logically (rather than physically) includes two parts: twig pattern matching (defined by FLW) and result construction (defined by Return).

Tree algebras have been developed to express more complex XQueries. [JAKC+02, PAKJ+02, PWLJ04] address transforming XQuery to an *algebraic tree*. The algebraic tree represents an efficient logical plan of answering XQuery. Each node in this tree is a tree algebraic operator. The basic tree algebraic operators are *selection*, *projection*, and *grouping*, each of which takes one or multiple twig patterns as inputs.

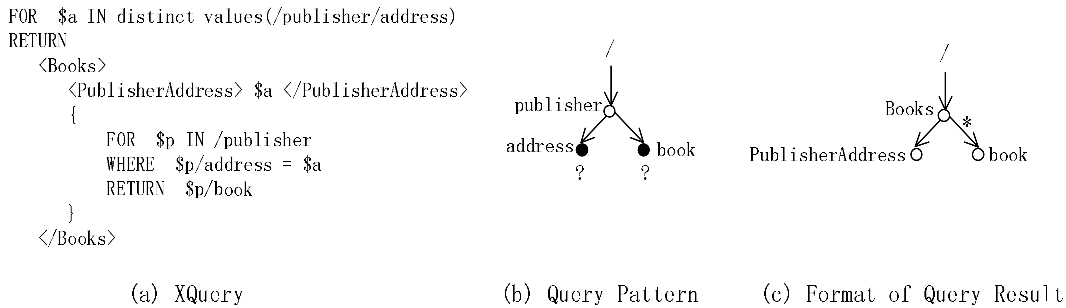


Figure 4: An example of XQuery

1.2.3 Summary

The core of both XPath and XQuery queries is *twig pattern matching* (also called *twig query*), i.e. finding from XML documents all matches that satisfy the twig (or path) pattern specified by a given query. We call nodes in XML data trees *data nodes* and nodes in query twigs *query nodes*. For XPath queries, the output of twig pattern matching is a set of *data nodes* whose corresponding query node is the end node of the *main path* in a query twig. For example, the output of matching the twig pattern in Figure 3 (b) is a set of *title* nodes. We call this type of output *single-node solutions*. For XQuery queries, the output of twig pattern matching is a set of *tuples* of data nodes that correspond to *multiple* query nodes in a query twig. For example, the output of matching the twig pattern in Figure 4 (b) is a set of (*address*, *book*) tuples, but not a set of only *book* or *address* nodes. We call this type of output *tuple solutions*.

Another important thing is that current XPath and XQuery do not support ID/IDREF axis queries, i.e. they always assume queries work on tree-shaped XML data model. In fact, this assumption has also been taken by most research papers on XML query processing. The first reason for taking this assumption is that general graph-shaped data model significantly increases the complexity of XML query processing. The second reason is that graph-shaped XML documents with ID/IDREF attributes are not usual in practical applications. So we will continue to take this assumption in this survey except explicitly claimed.

In the remainder of this survey we review major techniques for processing XML twig queries. We categorize these techniques into three classes based on the storage format of XML data. Section 3 introduces the *file approach*, in which XML data must be stored in commonly used *flat files*, as required by special-purpose applications. Sections 4 and 5 introduce the *relational approach* and the *native approach*, in which XML data are stored in *relational databases* and *inverted lists*, respectively. With value indexes and structural indexes available in these two approaches, XML queries can be answered much more efficiently than in the file approach. Before we begin to review these approaches, we first introduce *numbering schemes*.

2 Numbering Schemes

In this section, we introduce *numbering schemes* that can overcome the weakness of the file approach and have been taken as an important foundation for many techniques in the relational and native approach.

Edges in XML data trees represent structural relationships between data nodes. The key idea of answering XML twig queries is just determining structural relationships, or more specifically reachability, between any pair of nodes in XML data trees. For example, in order to answer a path query '$A//B$', given any pair of A-tagged node and B-tagged node, say (a, b), in a data tree, we need to determine whether there exists a

path from a to b.

A straightforward method of determining reachability is *tree navigation* [MW99], which consists of either traversing the subtree rooted at an A-tagged node to see if a B-tagged node can be found (*forward navigation*), or, more intelligently, backtracking from a B-tagged node upwards to see if an A-tagged node can be found (*backward navigation*). Backward navigation is usually more efficient because each node in a tree has only one incoming path from the root but multiple outgoing paths. However, if A-tagged nodes are more selective than B-tagged nodes, i.e. most A nodes have B descendants but most B nodes have no A ancestors, then forward navigation might be more efficient. Therefore, a trade-off has to be determined, which is just the motivation of hybrid navigation [MW99]. However, on the whole, the navigational method is not efficient, since both forward and backward navigations involve traversing a large amount of irrelevant nodes, i.e. nodes tagged with neither A nor B. For example, for a path '$/A/D/E/F/B$' in a data tree, irrelevant nodes tagged with D, E or F have also to be traversed for answering a query '$A//B$' when the navigational method is used.

Another method of determining reachability is precomputing, for each node in a data tree, a set of nodes that can be reached from this node, i.e. materializing transitive closure of this data tree. The transitive closure is typically very large and so could waste storage space. Therefore, we need a less exhaustive method to compactly represent transitive closure. *Numbering Schemes* is just one such method.

[Die82] is the origin of numbering schemes for trees. It proposed a kind of numbering scheme we call *PrePost Coding*, which uses tree-traversal orders of nodes to compactly represent transitive closure of trees. Specifically, each node in a tree is labelled with a pair of numbers, (*start*, *end*), where *start* and *end* correspond to *preorder* and *postorder* traversal numbers of this node in the tree, respectively. [ZND+01] introduced *PrePost* coding into XML applications. As can be seen from Figure 2 (a), the following property always holds.

Property 1 (Ancestor-Descendant Relationship) *In a data tree, node a is an ancestor of node b if and only if $a.start < b.start < a.end$.*

Obviously, *PrePost Coding* has two big advantages. (1) (start, end) numbers (also called *PrePost numbers*) only need modest storage space: $2 * |V|$, where $|V|$ is the number of nodes in the data tree. (2) Using PrePost numbers, we can efficiently determine the ancestor-descendant relationship between any pair of nodes in constant time by using only two *number comparison* operations. In addition, *PrePost coding* can also be easily extended to check the parent-child relationship if we attach another number, *level*, to each node, which denotes the depth of this node in tree.

Property 2 (Parent-Child Relationship) *In a data tree, node a is a parent of node b if and only if $a.start < b.start < a.end$ and $a.level + 1 = b.level$.*

In fact, in addition to commonly used '/' and '//' axes, *PrePost coding* extended with the number *level* is able to process all other axes defined in XPath, such as *following*, *following-sibling*, etc [Gru02, GvKT04].

Another famous numbering scheme for trees is *Dewey Coding* [OCL04], which was originally developed for general knowledge classification. [TVB+02] introduced it into XML query processing. With this coding, each node is associated with a *vector* of numbers that represents the path from the root to this node. This coding method is illustrated in Figure 5. We can show that in a data tree, node a is an ancestor of node b if and only if $a.vector$ is a prefix of $b.vector$.

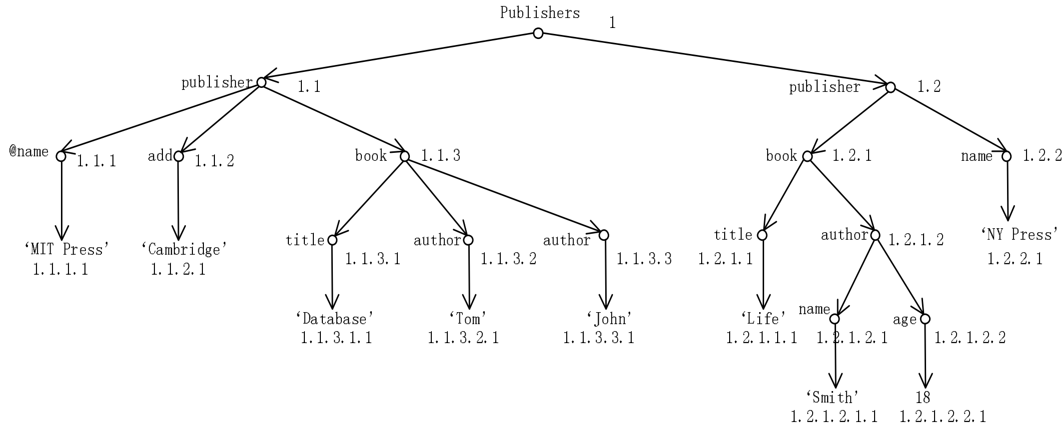


Figure 5: Dewey Coding

An advantage of Dewey Coding over PrePost Coding is that Dewey Coding is easier to maintain when dynamic updates occur on data trees. Using Dewey Coding, when a new node is inserted somewhere in a data tree, only nodes in subtrees rooted at the following sibling nodes of this new node need to change their Dewey vectors. In contrast, using PrePost Coding, when a new node is inserted, most nodes in a data tree might need to update their (*start*, *end*) numbers. *ORDPATH Coding*, which is a variant of Dewey Coding but even easier to maintain than Dewey coding, has been integrated into the XML query processing component of Microsoft SQL Server 2005 [OOP+04].

However, compared with PrePost, Dewey has some obvious weaknesses. (1) The path vector associated with each node needs more storage space than (start, end) numbers in PrePost Coding. (2) PrePost provides more efficient support in checking the ancestor-descendant relationship between two nodes, since *number comparison* operation can be implemented more efficiently than the operation of checking the *prefix containment* relationship between two path vectors. Due to the nice properties of PrePost, most XML research papers use PrePost as their numbering schemes. Our survey will continue this tradition.

In addition to numbering schemes for trees, numbering schemes have also been developed for DAGs [ABJ89] and for even more general graphs with cycles [CHKZ03]. [STW04, STW05] applied 2-hop labels developed in [CHKZ03] to deal with general XML data graphs. However, the size of 2-hop labels is usually very large, which limits its application in practice.

3 XML Query Processing: the File Approach

XML data are originally created in the form of XML documents (Figure 1) and stored in flat files. Generally, various indexes need to be built on XML data to facilitate answering XML queries, since indexes can locate goal data quickly without exhaustively scanning the data. Such indexes include classical B+-tree index (Section 4), which is an index on *data values* (*value indexing*), and recently developed numbering schemes (Section 2), which is an index on *structure* of XML documents (*structure indexing*). However, indexes themselves are *redundant* data. In some application scenarios, XML data must be exchanged in the form of flat files only, without any redundant data such as indexes being allowed to associate with them. In those cases where indexes are not available, entire XML documents have to be scanned to answer queries.

One example of such applications is SDI (Selective Dissemination of Information) [AF00, DFFT02, DF03,

DAF[+]03, BGKS03, TRP[+]04]. SDI is essentially an XML Publish/Subscribe system. Figure 6 illustrates its structure. The filtering system stores XPath queries from subscribers. It matches each incoming streaming XML document D from publishers with each subscribed XPath query. If a match is found in D with some XPath query Q, then D will be sent to subscribers of Q. In order to reduce network bandwidth, publishers disseminate only XML documents, without any redundant data such as indexes associated with these documents. In this scenario, only *tree navigation* methods, specifically only the *forward navigation* method (Section 2), can be used, since scanning XML documents sequentially in document order is essentially a *depth-first* traversal of XML data trees.

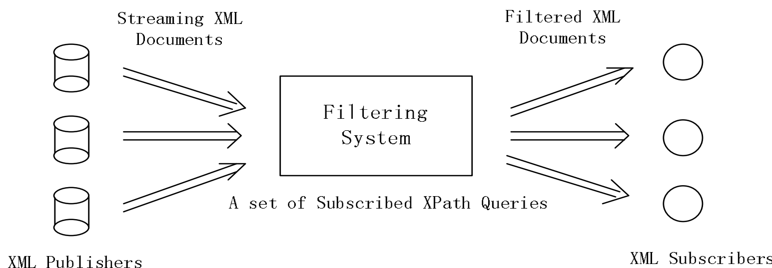


Figure 6: SDI application

3.1 Single-Query Processing

3.1.1 The Automata Approach

The *automata* approach is a natural implementation of forward navigation, which has been widely researched [AF00, DFFT02, DF03, DAF[+]03, BGKS03, HBG[+]03]. This approach expresses an XPath query as an automaton and runs XML documents on this automaton as if XML documents were strings.

When a streaming XML document arrives, SAX parser [Org04] parses it sequentially on the fly. *SAX* is an event-based XML parser. A *StartElement* event is triggered when the opening tag of an XML element is encountered, which returns the tag name and all associated attributes (if any) of this element to the event handler. Similarly, an *EndElement* event is triggered when the closing tag of an XML element is encountered, which returns the tag name of this element to the event handler. The event handler then uses opening/closing tags returned by events to activate corresponding state transmissions of automaton.

Figure 7 illustrates this approach. Figure 7 (b) is an automaton equivalent to XPath query '$//A//B/C$' where '//' axes are represented using $*$-edges ('$*$' denotes any tag name), and the leaf query node is taken as an *accept state* (State 3). The key idea of this approach is using a run-time stack, in which each stack element is a set of automaton states. When an opening tag is encountered, each state in the stack-top element is transformed to new states (or to this state itself if there is a $*$-edge outgoing from it) based on this tag. These newly generated states are collected into a new stack element which is in turn pushed into the run-time stack as the new stack top. Instead, when a closing tag is encountered, the stack-top element is simply popped out of the stack. For SDI applications whose goal is just to check if there exists *one* match between the published XML document and subscribed XPath queries, the matching process can terminate once an acceptable state, such as State 3 in Figure 7 (b), is reached. However, for general query applications whose goal is to find *all* matches, the matching process has to continue until the end of the XML document is reached. All elements resulting in accept states, such as elements c_1 and c_2 in Figure 7 (c), are output as query results.

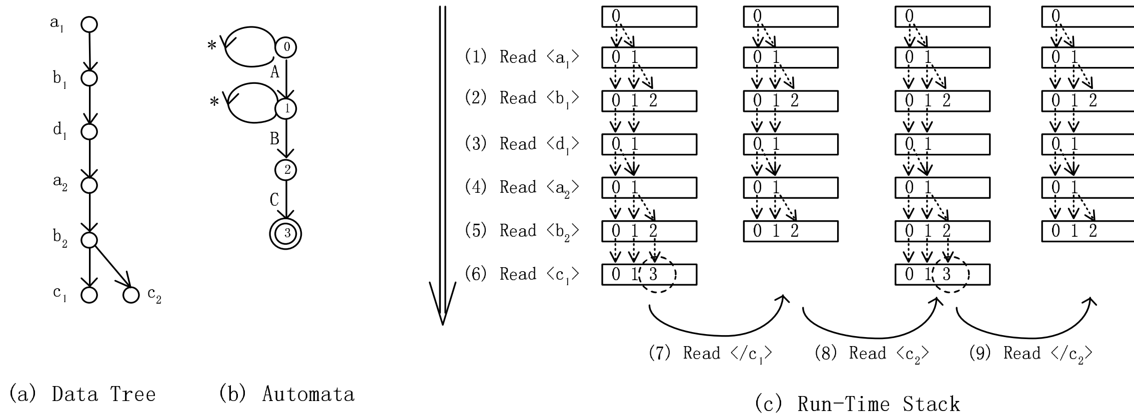


Figure 7: The Automata approach: processing XPath query '$//A//B/C$'

3.1.2 The PathStack Approach

The automata approach described above is simple and feasible. However, its big weakness is that although it derives *single-node solutions* (e.g. a set of C nodes), it is difficult to derive *tuple solutions* (e.g. a set of (A, B, C) tuples). The reason is that the run-time stack tracks only *states* in automata but not *data nodes* in data trees. In addition, the run-time stack wastes memory space. Due to the '//' axes, states with outgoing $*$-edges, such as State 0 and State 1, have copies in a large number of stack elements repeatedly.

[BKS02] introduced an elegant data structure, *PathStack*, which can overcome the weaknesses of the automata approach described above. *PathStack* was introduced in [BKS02] originally as a *native approach* to answering XML twig queries. [BGKS03] extended it to process multi-queries. Here we only introduce its role in the file approach while leaving the introduction to its role in the native approach to Section 5. Figure 8 illustrates this PathStack approach.

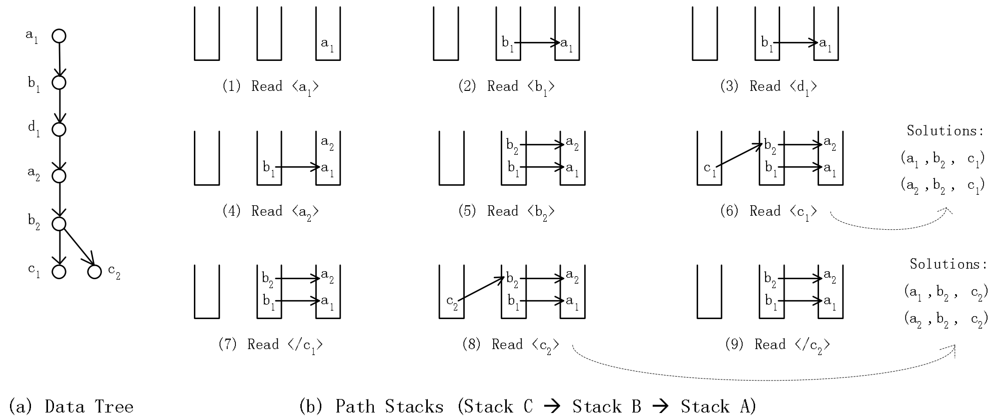


Figure 8: The PathStack approach: processing XPath Query '$//A//B/C$'

The key idea of the PathStack approach is using a series of linked stacks to track scanned data nodes. Specifically, one stack is created for each query node in a path query. For example, for a path query '$//A//B/C$', there are three stacks, Stack C → Stack B → Stack A. When an opening tag is encountered, the corresponding XML element is pushed into the stack named by this tag, associated with an a pointer to

the *top element* in its parent stack (see Steps (2) (5) (6) and (8)). Note that such elements as d_1 whose tags do not correspond to any stack (i.e. are irrelevant to the given path query) are simply discarded. Instead, when a closing tag is encountered, the top element in its corresponding stack is simply popped out (see Steps (7) and (9)).

The procedure above guarantees that at all times, elements in all stacks are from the same path in the data tree. Therefore, when an element is pushed into the stack corresponding to the end node of the path query, such as Stack C, it implies that some matches might have been found. These matches can be output immediately as solutions through backtracking pointers associated with the elements in stacks (see Steps (6) and (8)). Note that in order to check the child-parent relationship, each element that is pushed into the stack also needs to be associated with its *depth* number in the data tree, which can be easily derived in the parsing process.

Compared with the automata approach, the PathStack approach has the following advantages. (1) The PathStack approach saves memory space. The number of stacks in the PathStack approach is the length of the path query, while the depth of the run-time stack in the automata approach is the depth of the entire XML data tree. (2) More importantly, the PathStack approach can derive tuple solutions, rather than only single-node solutions. Tuple solutions are very important. They might be required by XQueries (Section 1.2.3). More importantly, tuple solutions help answer *twig* queries as well as *simple path* queries through using a post-joining procedure as introduced below.

3.1.3 The TwigStack Approach

The TwigStack approach extends the PathStack approach to answer general *twig* queries [BKS02]. Its key idea is *twig decomposition*, i.e. decomposing twig queries into multiple root-to-leaf path queries. Each path query is still processed as in PathStack, and the query results are finally joined together to get the result of the original twig query. However, since path queries from the twig decomposition have common prefix (query) nodes, the stacks corresponding to these common prefix nodes can be shared. Therefore, in contrast to PathStack, which links stacks in the form of a path, TwigStack links stacks in the form of a twig.

An example of TwigStack is shown in Figure 9. In this example, once a C-tagged node is pushed into Stack C, the tuple solutions obtained from Stack C → Stack B → Stack A are immediately sent to Table 1. Similarly, once a E-tagged node is pushed into Stack E, the tuple solutions obtained from Stack E → Stack D → Stack B → Stack A are immediately sent to Table 2. If a path query is very selective so the size of Table 1 and Table 2 is small, then these two tables can be temporarily stored in memory. Otherwise, they have to be sent to disk. Finally, after the entire XML document is scanned, there is a post-joining procedure which joins Table 1 and Table 2 on their shared attributes, A and B, to get tuple solutions (A, B, C, D, E).

3.2 Multi-Query Processing

The problem of *multi-query processing* is answering a batch of queries rather than a single query. For example, the SDI application is essentially an XML multi-query processing problem except that it is only to find *one* rather than *all* matches of a streaming XML document with each of subscribed queries.

The problem of multi-query processing has been widely researched in the context of relational databases (e.g. in [RSSB00]). The key idea of improving the performance of multi-query processing is answering multiple queries simultaneously rather than separately through exploring shared parts of these queries. This idea is similarly applicable in the context of XML multi-query processing.

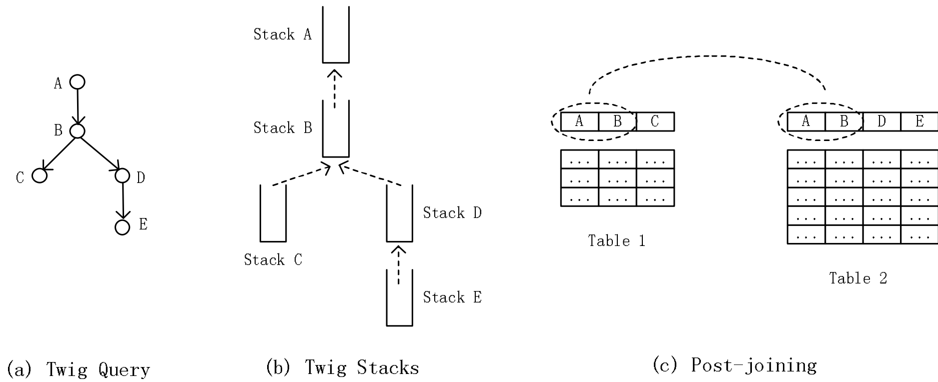


Figure 9: The TwigStack approach

It is straightforward to extend three approaches to XML single-query processing (Section 3.1) to XML multi-query processing. The extension of the automata approach finds common prefixes of the given path queries and share the states corresponding to these common prefixes in a newly constructed automaton [DFFT02, DF03, DAF+03], as Figure 10 (a) illustrates. Similarly, the extension of the PathStack approach finds common prefixes of the given path queries and share the stacks corresponding to these common prefixes in a newly constructed TwigStack [BGKS03], as Figure 10 (b) illustrates. Note that the extension to the PathStack approach above is essentially the same as the TwigStack approach.

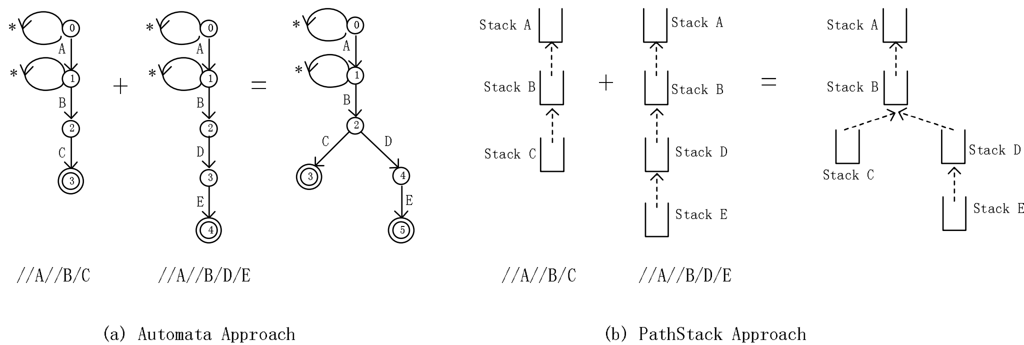


Figure 10: XML Multi-Query Processing

Through the extensions above, each XML document needs to be scanned only once to answer multiple queries simultaneously.

3.3 Summary

The file approach is mainly used for special-purpose applications in which XML data *must* be stored in commonly used flat files in the form of just original XML documents. Because no redundant data such as indexes are available in such applications, entire XML documents have to be scanned sequentially element by element despite the fact that most elements in documents might be irrelevant to the specified queries, which usually results in poor query processing performance. In the following two sections, we investigate the relational approach and the native approach, in which XML data are stored in relational databases and in inverted lists, respectively. With value indexes and structural indexes available in these two approaches, XML queries can be answered much more efficiently than in the file approach.

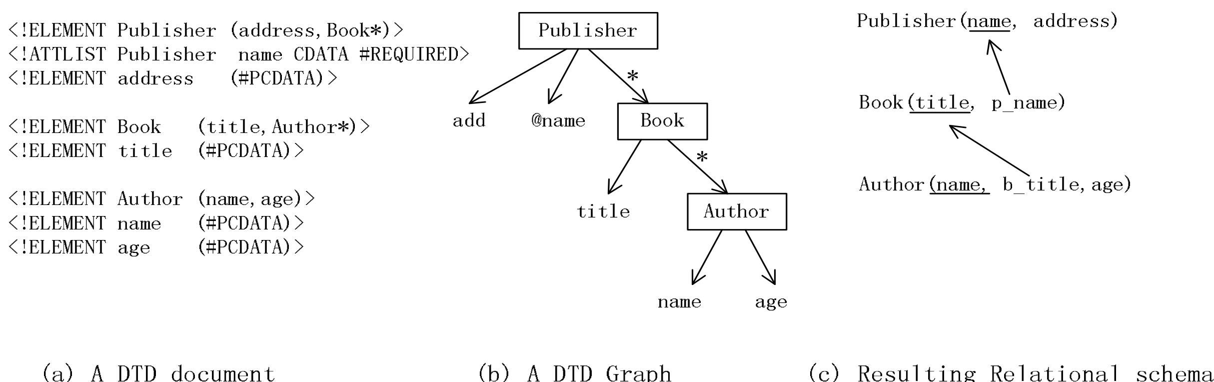
4 XML Query Processing: the Relational Approach

Relational database systems are today's mainstream database systems. Today's well-known commercial database systems, such as IBM DB2, Microsoft SQL Server, and Oracle, are all relational database management systems (RDBMS). Due to more than thirty years of academic and industrial efforts, RDBMSs have acquired strong capabilities in storage management, query processing and optimization, concurrency control and recovery, etc. Therefore, a lot of research efforts have addressed storing and querying XML data in RDBMS. In Section 4.1, we review past work on storing and querying XML data with a 'schema'. In Sections 4.2 through 4.4, we review past work on storing and querying schemaless XML data.

4.1 The DTD Approach

This approach is developed to store and query XML data with a 'schema'. As introduced in Section 1, XML is a flexible data model. However, XML data in many practical applications also conform to a schema to some extent, since for various inter-operating applications that exchange data with each other, a common agreement on the schema of exchanged data will facilitate data exchange among them significantly. Such schemas can be described using standard Document Type Descriptors (DTDs) [Gro04b] or XML Schemas [Gro04d]. Here we briefly introduce basic issues on DTD only, since XML Schemas are essentially extensions to DTDs.

DTD is a set of statements where each statement specifies a relationship between an XML element and its sub-elements/attributes, or the data type of an XML element/attribute. DTD statements are usually stored in a special document for reference. If an XML document, X, cites a DTD document, D, on its file head, then the structure of XML data in X must conform to the schema specified by D. We show a simple DTD example in Figure 11. Figure 11 (a) is a DTD document, whose semantics can also be explained using a DTD graph in Figure 11 (b). The '$*$' symbol associated with an element in DTD statements implies that this element can have multiple copies under its parent element. For example, a *Publisher* element might have multiple *Book* sub-elements.



(a) A DTD document (b) A DTD Graph (c) Resulting Relational schema

Figure 11: An DTD example

DTD schemas can be naturally transformed into relational schemas [STZ^{+}99, SSK^{+}01], as Figure 11 (c) illustrates. In the resulting relational schema, separate relations are created for the root element (Publisher) and all '$*$' sub-elements (Book and Author) in DTD. Each '$*$'-element relation has a foreign-key reference, e.g. attribute *p_name* in the *Book* table and attribute *b_title* in the *Author* table, to its parent-element table. After XML data conforming to a DTD schema have been shredded into relational tables, XML queries over

XML data can be easily transformed into SQL queries over relational data. For example, a twig query $/Publisher[address = `Cambridge']//Author/name$ can be transformed into a SQL query that joins three tables, Publisher, Book, and Author, together, as Figure 12 illustrates.

```
Select a.name
From   Publisher p, Book b, Author a
Where  p.address = 'Cambridge' and
       p.name = b.p_name  and  b.title = a.b_title
```

Figure 12: The DTD approach: SQL query for '$/Publisher[address = `Cambridge']//Author/name$'

4.2 The Edge Approach

- **The Basic Edge Approach**

[FK99] proposed a simple approach to shredding *schemaless* XML data into relations. This approach is based on *edge-labeled* XML data trees. In this approach, all edges in a data tree are stored in a single relational table, *Edge*. The schema of this *Edge* table is shown in Figure 13. The key idea of this schema is an attribute pair (*Source*, *Target*), which represents end points of edges. Attribute *Label* represents tags on edges. Attributes *Flag* and *Value* give the type and value of target nodes of edges, respectively. As an example, Figure 13 populates the *Edge* table with XML data shown in Figure 2 (b).

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
...	...	...	...	...

Figure 13: The Edge Table

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 twig queries without "//" axes into SQL queries. The transformation method is illustrated in Figure 14 with a twig query '$/Publisher[address = `Cambridge']/book/author/name$'. Execution of this SQL query comprises two steps. The first step is a *candidate-edge finding* step, which retrieves data edges for each label in the twig query, as Part (1) in Figure 14 shows. We can see that a clustered index pre-built on the*Label* attribute can significantly speed up the processing of this step. The second step is an *edge joining* step, which joins adjacent edges as Part (2) in Figure 14 shows. The processing of this step can be made more efficient by pre-building indexes on attributes ($Source$, $Target$).

- **The Binary Approach**

A weakness of the above Edge approach is that it involves multiple self-joins of the large Edge table. For example, five Edge tables are joined in Figure 14, one table for each query node in the query twig. In order to overcome this weakness, [FK99] also proposed a *Binary* approach, which is a variant of the basic Edge approach, to avoid exploring the large Edge table. The key idea of this approach is grouping all edges with the same label into one table respectively, i.e. creating one table for each distinct label. Each label table has the schema (Source, Target, Flag, Value), with the *Label* attribute being dropped from the Edge schema. An example of a SQL query against this schema is shown in Figure 15. In this example, the candidate-edge finding operations in Part (1) of Figure 14 are saved. In addition to improving query processing performance,

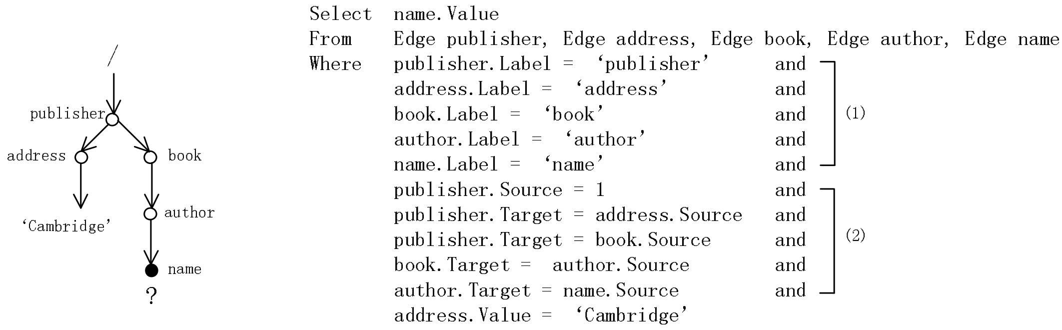


Figure 14: The Edge approach: SQL for '$/Publisher[address = `Cambridge']/book/author/name$'

the Binary approach also saves storage space, since it doesn't store *labels* of edges. However, for large XML documents with a lot of distinct labels, the Binary approach will unavoidably result in a large number of relational tables, which increases the management workload of DBMS. Otherwise, we notice that the basic idea of this approach that clusters edges by their labels is very similar to the idea of inverted lists that will be introduced in Section 5.

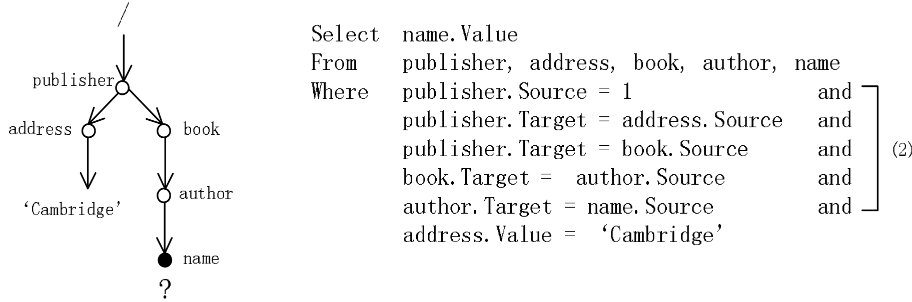


Figure 15: The Binary approach: SQL for '$/Publisher[address = `Cambridge']/book/author/name$'

In a whole, the Edge approach has two weaknesses. (1) It involves many join operations. The number of joins is just the number of query nodes in a twig query. So it fails to process large twig queries efficiently. (2) Its biggest weakness is that it does not support twig queries with "//" axes (e.g. '$A//B$'), since it does not know how many tags and which tags are involved between tag A and tag B.

4.3 The Node Approach

As we introduced in Section 2, *numbering schemes* are essentially *structural indexes*, which help answer '//' axis queries efficiently. [ZND$^+$01] is the first paper that applied PrePost coding developed in [Die82] to XML research. This paper contributed a *Node approach* to shredding *schemaless* XML data into relations. This approach is based on *node-labeled* XML data trees. In this approach, all *internal nodes* (i.e. element nodes and attribute nodes) in a data tree are stored in a relational table, *Node*. The schema of this *Node* table is shown in Figure 16. The key idea of this schema is an attribute triple (*Start*, *End*, *Level*), which replaces the attribute pair (*Source*, $Target$) in the *Edge* schema. '//' axis queries can be answered efficiently through using (*start*, *end*) numbers of nodes. *Level* is used with (*start*, *end*) together to answer '/' axis queries. As an example, Figure 16 populates the *Node* table with XML data shown in Figure 2 (a).

Based on Property 1 and Property 2 in Section 2, it is easy to transform XML queries with both '/'

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
...	...	...	...	...	...

Figure 16: The Node Table

and ‘//’ axes into SQL queries. The transformation method is illustrated in Figure 17 with a twig query ‘$/Publisher[address = `Cambridge']//author/name$’. Similar to the *Edge* approach, execution of this SQL query comprises two steps, *candidate-node finding* (Part (1)) and *node joining* (Part (2)). The difference is that in the second step, the *Node* approach joins nodes using (*Start*, *End*, *Level*) attributes. Just as in the *Edge* approach, Part (1) can be saved in the *Node* approach if a variant similar to the Binary approach is used.

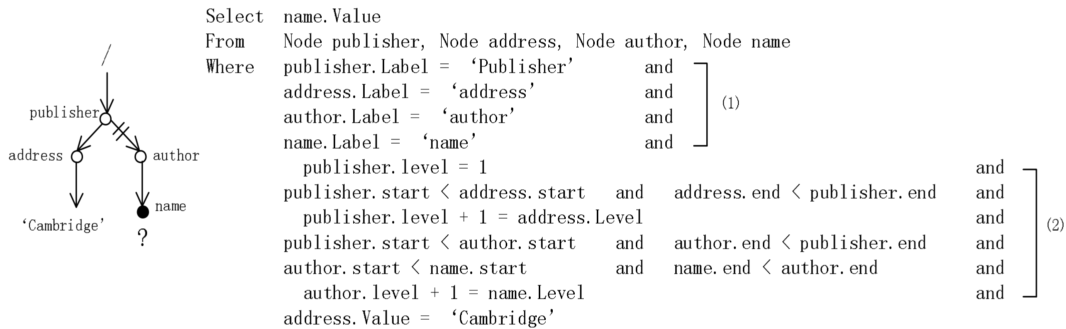


Figure 17: The Node approach: SQL query for ‘$/Publisher[address = `Cambridge']//author/name$’

The *Node* approach overcomes the weakness of the *Edge* approach which does not support‘//’ axis queries. However, similar to the *Edge* approach, it involves many join operations. Specifically, the number of joins is just the number of query nodes in a twig query, which results in inefficient query processing of large twig queries.

4.4 The Path Materialization Approach

• The Basic PM Approach

In order to reduce the number of node joins, [YASU01] proposed a Path Materialization (PM) approach to shredding *schemaless* XML data into a relation table, *Path*. The schema of this *Path* table is shown in Figure 18. It 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 new attribute *Path*.

Through the *Path* attribute, the PM approach can answer twig queries efficiently in units of *paths* rather than in units of single *edges*. Specifically, given a twig query, the PM approach first decomposes it into multiple root-to-leaf path queries as the TwigStack approach in Section 3.1.3 does, and then joins results of these paths queries together. Figure 19 illustrates how to use a SQL query to answer a twig query ‘$/Publisher[address = `Cambridge']/book/author/name$’. Part (1) is the *twig decomposition* step, which

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

Figure 18: The Path Table

uses the value of *root paths* of leaf nodes (*address*, *name*) and branching nodes (*publisher*) in the query twig to retrieve their corresponding data nodes in data tree. Part (2) is the *path joining* step, which joins data nodes retrieved from Part (1) through their (start, end) numbers.

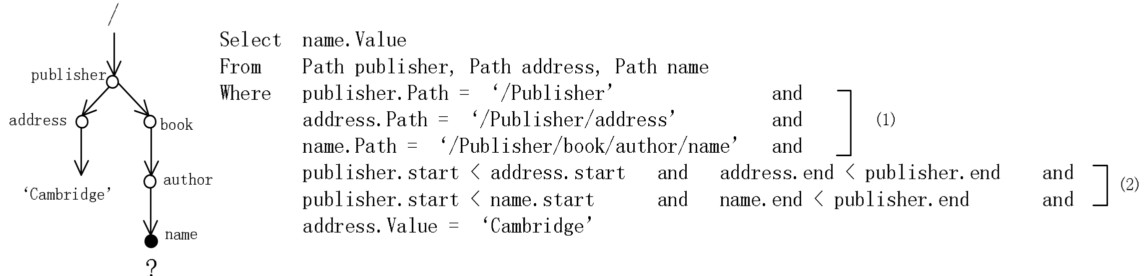


Figure 19: The Basic PM approach: SQL for '$/Publisher[address = `Cambridge']/book/Author/name$'

The PM approach has two advantages. (1) It involves fewer join operations in Part (2) than the Node approach, since it answers twig queries in units of *paths* rather than in units of single *edges*. For example, for the twig query in Figure 19, the Node approach needs to join five Node tables but the PM approach needs to join only three Path tables. Therefore, the PM approach generally has higher query processing performance. (2) The PM approach can also support '//' axis queries as the Node approach does, by using the Optional String Pattern Matching (OSPM) function ("LIKE") provided by SQL. For example, in order to answer a query "$/Publisher[address = `Cambridge']//name$", we only need to replace "name.Path='/Publisher/book/author/name'" in the *where* clause in Figure 19 with "name.Path LIKE '$/Publisher/\%/name$'".

However, we can also observe that although the number of *join* operations in Part (2) is reduced, it is at the expense of increasing the complexity of *selection* operations in Part (1). As we know, SQL supports Exact String Matching ("=") efficiently through pre-building a B+-index on string attributes, but B+-indexes do not support optional string pattern matching ("LIKE") efficiently due to the inherent structure of B+-trees. In order to find patterns with multiple '%' symbols, a large number of irrelevant strings in tables might have to be checked exhaustively. Therefore, the PM approach does not support '//' axis queries efficiently when there are multiple '//' axes in queries (e.g. $//A//B/C//D$).

- **The RP Approach**

[PCS+04] proposed a Reversed Path (RP) approach to overcome the weakness of the PM approach discussed above. This approach uses a schema shown in Figure 20. Its key idea is storing *reversed* root paths of data nodes in a new attribute *ReversedPath*. Otherwise, the RP approach uses an $ORDPATH$ attribute to replace the $(start, end)$ attribute pair in the PM approach. *ORDPATH* coding is a variant of *Dewey* coding we mentioned in Section 2. It can be used to determine ancestor-descendant/parent-child relationships

between nodes as *PrePost* coding does [OOP+04]. Here we simply ignore the difference between ORDPATH numbers and (Start, End) numbers, and concentrate our discussion on the *ReversedPath* attribute.

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
...	...	...	...

Figure 20: The ReversedPath Table

Figure 21 shows an example of how the RP approach answers twig queries with multiple '//' axes. The first step is still *twig decomposition*, which decomposes the query twig into three paths. However, Path (3) involves three '//' axes. In the PM approach, we have to use '$\%A/B/C\%E/F\%G$' as a search pattern on the *Path* attribute to retrieve corresponding data nodes. As we analyzed earlier, this is not efficient. Therefore, the RP approach continues to decompose Path (3) into Path (4) and Path (5), both of which include only one '//' axis just in the beginning. So we can use '$/F/E\%$' and '$/G\%$' as search patterns on the *ReversedPath* attribute to retrieve data nodes of Path (4) and Path (5), respectively. So the task here is just finding a string with a specified *prefix*, which can be implemented more efficiently than the general Optional String Pattern Matching task with multiple '%' symbols. Finally, similar to the PM approach, the RP approach has a *path joining* step, which joins results of path queries together through the ORDPATH attribute.

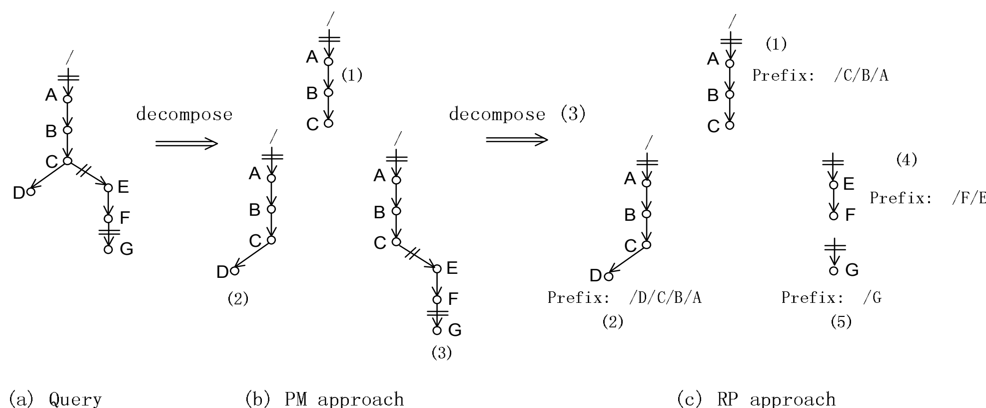


Figure 21: The RP approach

- **The BLAS Approach**

As we saw above, the RP approach in [PCS+04] has simplified the task of general Optional String Pattern Matching to an easier task of String Prefix Matching (SPM). However, [PCS+04] did not provide any details on how to efficiently implement SPM. It seems that they just simply push the SPM task down to the SQL engine. In contrast, another work [CDZ04] not only introduced the RP approach independently from [PCS+04] but also developed a very intelligent method named *BLAS* (Bi-LAbeling System) to implement SPM efficiently. The key idea of BLAS is encoding each ReversedPath string into a number, *PLabel*. This encoding method is illustrated in Figure 22.

In this example, we assume that there is a total of four distinct tag names in some XML document, $p1$ through $p4$. At the first level, these four tags divide reserved number space [0, 1024) into four equal-length

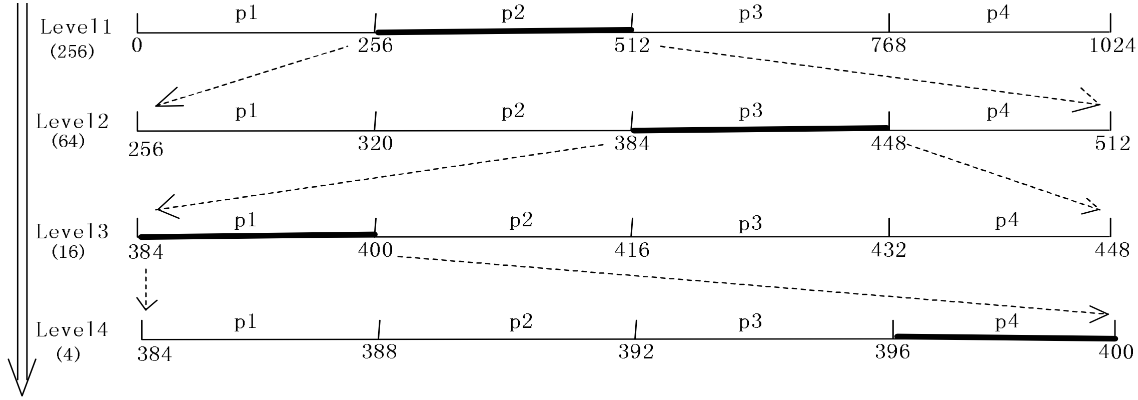


Figure 22: How to compute PLabel('/p2/p3/p1/p4')

segments, each with length $1024/4 = 256$. In the same way, at the second level, four tags divide *each* segment at the first level into four equal-length segments, each with length $256/4 = 64$, and so on so forth. So we have

$$Plabel(`/p2/p3/p1/p4') = 256*(2-1) + 64*(3-1) + 16*(1-1) + 4*(4-1) = 396$$

In the same way, we can also get

$$PLabel(`/p4/p2/p3') = 256*(4-1) + 64*(2-1) + 16*(3-1) = 864$$

A very nice property of Plabel is that all strings with common prefixes cluster in adjacent digital areas. For example, all ReservedPath strings with prefix '/p2/p3' cluster together. So if we pre-build a clustered B+-tree index on the Plabel attribute of the *ReversedPath* table, then all reversed paths with the specified prefix can be retrieved very efficiently using a SQL *range query*. For example, in order to retrieve all reversed paths with prefix '/p2/p3/', BLAS first computes $lower_bound = PLabel(`/p2/p3/') = 384$ and $higher_bound = PLabel(`/p2/p4/') = 448$. Then a SQL range query is issued to retrieve all reversed paths with PLabel within $[384, 448)$.

4.5 Summary

In this section, we saw that XML data can be simply loaded into relational databases and XML twig queries over XML data can also be easily transformed into SQL queries over relational data. In the relational approach, all query processing work is pushed into *relational query optimizer* and no extra processing work is needed.

When XML data conform to a schema such as DTD, the DTD approach introduced in Section 4.1 provides better query processing performance than other approaches introduced in Sections 4.2 through 4.4. The reason is that the DTD approach generates different relational schemas for different DTDs. Each generated relational schema is tailored for a specific DTD and so precisely captures the structure of XML data conforming to that DTD schema. In contrast, approaches in Sections 4.2 through 4.4 generate the same relational schema (tables *Edge*, *Node*, *Path*, etc) for various XML data despite their different structures, and so fail to efficiently process a specific goal data set. The experimental work in [TDCZ02] also verifies this point.

When XML data is schemaless (i.e. a DTD for it is not available), the PM approach is the best compared with the *Edge* approach and the *Node* approach, since (1) it supports '//' axis queries and (2) it needs fewer join operations. Further, among the three variations of the PM approach (Basic PM, RP, BLAS), the RP approach with the BLAS extension is the best. In fact, the basic RP approach has been integrated into Microsoft SQL Server 2005 [OOP$^+$04, PCS$^+$04]. Interestingly, [OOP$^+$04, PCS$^+$04] do not mention the work of BLAS [CDZ04]. We propose to extend the basic RP approach in [OOP$^+$04, PCS$^+$04] with the PLabelling method in BLAS to gain the best query processing performance.

5 XML Query Processing: the Native Approach

Although the relational approach is simple and feasible, it could have inferior query performance. In order to answer '//' axis queries, the *Node* approach and the *PM* approach use θ-joins [2] to implement *node/path-joining* step (see Part (2) in Figures 17 and 19), discarding equi-joins used in the *Edge* approach (see Part (2) in Figure 14). θ-joins are more complex and costly than equi-joins. Although current DBMSs have been coupled with efficient techniques to optimize and process equi-joins, they do not support θ-joins efficiently, particularly when multiple comparison predicates are involved in queries. Some experimental work has verified this point [ZND$^+$01].

Much research has been done on developing native algorithms to efficiently process θ-joins involved in XML twig queries. We say these techniques are in the *native* approach since their storage and query mechanisms are developed from scratch, without involving relational databases. The authors of these native techniques believe that a special storage and query system tailored for XML data will improve XML query processing performance significantly. In the native approach, θ-joins are also called *structural join*.

Specifically, in the native approach, XML data are stored in *inverted lists*. Inverted indexes have been widely used in *Information Retrieval* to implement efficient text search [SM83]. Inverted index creates one list for each distinct word in text documents; the list gives positions of all occurences of this word. These lists are called *inverted lists*. Borrowing this idea, the native approach creates one inverted list for each distinct tag in XML documents; the list gives positions of all elements with that tag name. Location of an element is expressed using its (start, end, level) numbers. Locations in a list are sorted in the increasing order of their *start* numbers. Figure 23 shows inverted lists of the XML document in Figure 2 (a).

<publisher> ⟶ (2, 20, 1), (21, 38, 1)

<book> ⟶ (9, 19, 2), (22, 34, 2)

<author> ⟶ (13, 15, 3), (16, 18, 3), (26, 33, 3)

<...> ⟶ ...

Figure 23: Inverted lists

5.1 The MPMGJN Approach

[ZND$^+$01] proposed an MPMGJN (Multi-Predicate MerGe JoiN) algorithm, which is the first native approach to implementing structural joins. Its implementation is somewhat similar to that of the standard *Merge Join*

[2] θ-joins are joins involving '$>$' and '$<$' comparisons, while equi-joins involve only '$=$' comparison.

algorithm developed in relational query optimizers for equi-joins. In order to answer a query '$A//B$' or 'A/B', two cursors are created on $AList$ and $BList$ that have been sorted in the increasing order of *start* numbers. Initially, these two cursors are pointing to the heads of $AList$ and $BList$, respectively. Then, they are compared with each other and advanced in line to implement merge join.

In contrast to the standard merge-join implementation for equi-joins, MPMGJN has its own cursor-advancing mechanism, which is specially tailored to efficiently support structural joins. Specifically, at each step, it compares and advances two cursors as Figure 24 describes. The working process of MPMGJN is also illustrated in Figure 25. Note that dotted edges in Figure 25 (a) mean there might be other data nodes than A-tagged or B-tagged nodes on those edges although we show only A-tagged and B-tagged nodes in this data tree. Experimental work in [ZND+01] found that MPMGJN algorithm is more than an order of magnitude faster than RDBMS join implementation in most query cases.

```
If  BList[cursorB].start < AList[cursorA].start  Then
    // advance the cursor of BList
    cursorB ++;
Else
    // begin the inner-loop join, then advance the cursor of Alist
    temp_cursorB = cursorB;
    while( BList[temp_cursorB].start < AList[cursorA].end  )
    {
        Output a tuple solution into the join result. Specifically,
            Case 1 (For the `A/B' query):
                   Output tuple (AList[cursorA], BList[temp_cursorB]) if
                   AList[cursorA].Level + 1 = BList[temp_cursorB].Level.
            Case 2 (For the `A//B' query):
                   Directly output tuple (AList[cursorA], BList[temp_cursorB]).
        temp_cursorB ++;
     }
     cursorA ++;
Endif
```

Figure 24: The core of the MPMGJN algorithm

5.2 The StackTree Approach

[AKJK+02] observed that although the MPMGJN approach is efficient for '//' axis queries, it fails to process '/' axis queries efficiently in some cases. A motivating example is shown in Figure 26 (a). In this example, a_1 has only two B children, b_1 and b_6. However, we can see from Figure 26 (b) that MPMGJN finds the child d_6 only after it has scanned b_1 through b_5, where b_2 through b_5, which are indirect descendants but not children of a_1, have to be visited unnecessarily.

In order to avoid such unnecessary node scanning, [AKJK+02] proposed a new approach, StackTree. StackTree uses a nice stack structure to cache A nodes nested on the same path in data trees. Figure 27 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, it is pushed into the stack. If it is a B-tagged node, StackTree tries to use it to form tuple solutions with existing A-tagged nodes in the stack. Figure 26 (c) illustrates its working process. From this example, we can see that there are no redundant comparisons of b_2 through b_5 with a_1. Therefore, StackTree has better query processing performance than MPMGJN.

Both StackTree and MPMGJN are *binary* join algorithms, i.e. they join only a pair of inverted lists (or only one edge in the query twig). Since a complete twig query consists of a series of binary joins, the problem of *join order selection* has to be considered seriously. Just as in the context of relational databases, join order

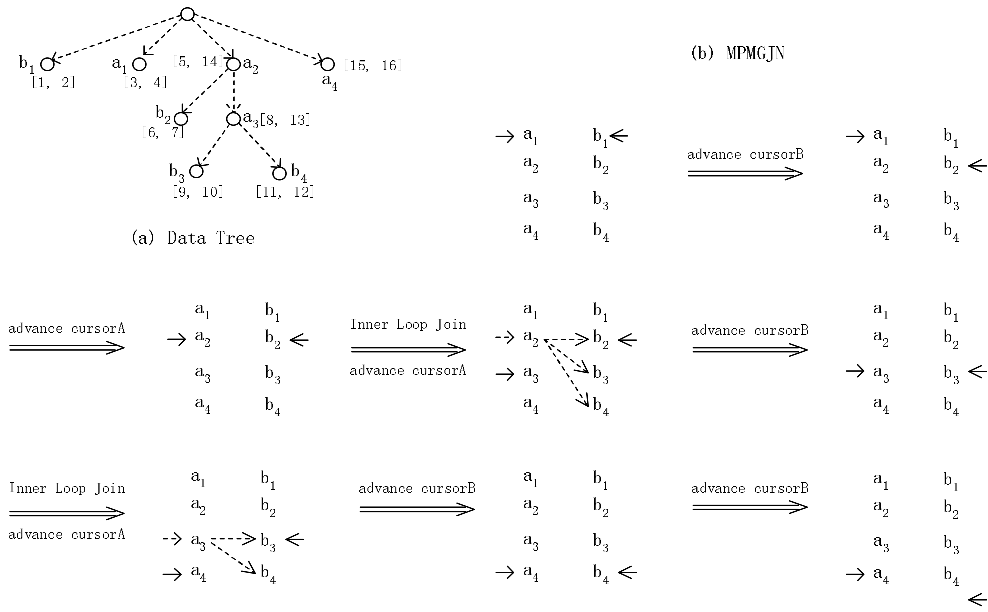


Figure 25: The MPMGJN approach (For the query '$A//B$')

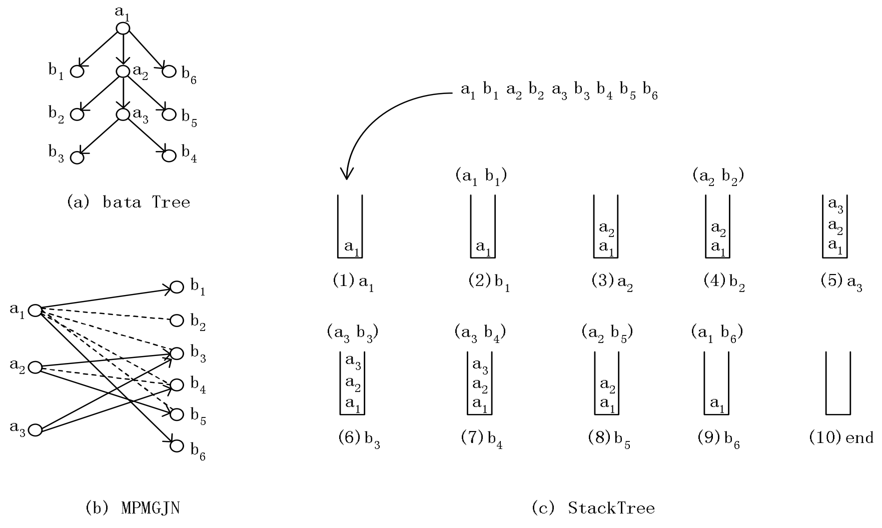


Figure 26: The StackTree approach (For the query 'A/B')

significantly affects XML query processing performance. As we know, most relational query optimizers use a classical *dynamic programming* method to select an optimal join order. [WPJ03] also proposed similar dynamic programming methods to select an optimal or sub-optimal order of binary structural joins for XML twig queries. The StackTree binary join algorithm and the corresponding dynamic-programming-based join order selection algorithm have been integrated into Timber [JAKC+02], a famous native XML database prototype from the University of Michigan.

```
min_start = Min(AList[cursorA].start, BList[cursorB].start)

Clear stack using min_start, i.e. all A nodes in stack with end number smaller
than min_start are popped out of stack.

If  min_start = AList[cursorA].start  Then
    Push node AList[cursorA] into stack;
    cursorA ++;
Else
    Output tuple solutions into the join result. Specifically,
        Case 1 (For the `A/B' query):
            Output tuple (top, BList[cursorB]), where top is the node on the
            top of the current stack and top.Level + 1 = BList[cursorB].Level.
        Case 2 (For the `A//B' query):
            Output all (a, BList[cursorB]) tuples, where a is any A node in the
            current stack;
    cursorB ++;
Endif
```

Figure 27: The core of the StackTree algorithm

```
min_start = Min_i{ List_i[cursor_i].start }

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.

Let min be id of the list such that List_min[cursor_min].start = min_start

If  min is not the id of leaf node of the path query  Then
    Push node List_min[cursor_min] into Stack_min with an associated pointer
    to the node on the top of its parent stack;
Else
    Output all tuple solutions implied by current stacks into the join
    result.
Endif

cursor_min ++;
```

Figure 28: The core of the PathStack algorithm

Another important thing is that the StackTree algorithm in Figure 27 outputs all tuple solutions in the increasing order of start numbers of descendant nodes (i.e. B-tagged nodes). For example, six tuple solutions in Figure 26 (c) are output in the order of b_1 through b_6. Complementarily, [AKJK+02] also proposed a variant of the StackTree algorithm to output tuple solutions in the increasing order of start numbers of ancestor nodes (i.e. A-tagged nodes). This is very important for twig queries. Consider a twig query '$C//A//B$'. If we select a query plan $C \bowtie (A \bowtie B)$, then query results of $A \bowtie B$ have to be sorted by A nodes, since the next binary join will occur between C and A.

Otherwise, [CVZ$^+$02] extends the StackTree algorithm with a *skip* technique so that some nodes in inverted lists do not need to be visited during the join process if these nodes are predicted not to form any tuple solutions with other nodes.

5.3 The Holistic Approach

A weakness of decomposing twig queries into multiple binary joins is that this method generates a large amount of intermediate query results. For example, for a query plan $(A \bowtie B) \bowtie C$, the query result of the first join $A \bowtie B$ has to be written to disk first if its size is too large to be contained in memory, and then be read back to memory to join with C after $A \bowtie B$ has been completed. This will result in high disk I/O cost. In order to overcome this weakness, [BKS02] proposed a Holistic approach, which is essentially a pipelining join, i.e. joining multiple inverted lists at one time so that no intermediate query results are generated.

Figure 28 shows the core of the PathStack algorithm which uses the Holistic approach to answer *simple path* queries. It is easy to see that this algorithm is structurally very similar to the StackTree algorithm in Figure 27. The difference is that StackTree uses only one stack to cache nested A nodes. In contrast, PathStack has multiple stacks, one for each non-leaf node in a path query, since inverted lists of all nodes in a path query are involved in pipelining joins. Also, each node cached in a path stack has an associated pointer to a corresponding node in its parent stack, in order to track tuple solutions.

Recall that PathStack was also used as a file approach to answering path queries over XML *documents* (Section 3.1.2). Here, Figure 29 illustrates how to use PathStack as a native approach to answering path queries over *inverted lists*. This figure is very similar to Figure 8. The differences are: (1) only A-tagged and B-tagged nodes are read in the native approach. Therefore, no other irrelevant nodes in XML documents, such as node d_1 in Figure 8, are read. (2) In the native approach, the event of nodes being popped out of stacks is triggered by the arrival of other nodes with higher start numbers than their end numbers, rather than being triggered by the arrival of their own closing tags as in the file approach.

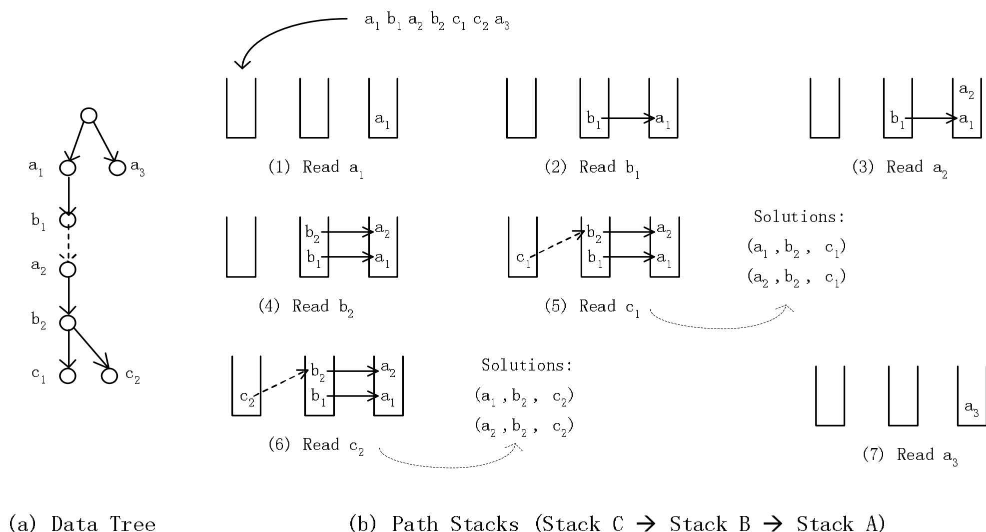


Figure 29: The Holistic approach (For the query '$A//B/C$')

Similarly, the Holistic approach also provides a TwigStack algorithm to answer general *twig* queries. The main idea of TwigStack has been illustrated in Figure 9.

[BKS02] also experimentally compared the Holistic approach with the StackTree approach. Their experimental results show that generally the Holistic approach has more than six-fold faster query processing performance than the StackTree approach coupled with the optimal join order. Due to its high query processing performance and algorithmic simplicity, the Holistic approach has been used extensively in some recent research work. For example, [JWLY03] extended it with a *skip* technique to avoid visiting some nodes in inverted lists that do not form any tuple solutions with other nodes. [JLW04] extended Holistic to process twig queries with OR predicates. [BGKS03] applied Holistic for multi-query processing.

6 Conclusions

In this survey we reviewed major techniques for processing XML twig queries. These techniques are categorized into three classes based on the storage format of XML data.

The file approach is mainly used for special-purpose applications in which XML data must be stored in commonly used flat files in the form of just original XML documents. Since no indexes are available in such applications, the entire XML document, including a large volume of elements irrelevant to the specified query, has to be visited, which usually results in poor query processing performance.

In the relational approach, XML data can be simply loaded into relational databases and XML twig queries over XML data can be easily transformed into SQL queries over relational data. In this approach, all specific query processing work is pushed into relational query optimizers and no extra processing is needed. However, current RDBMSs do not support θ-joins efficiently, despite the fact that θ-joins is a necessary component for answering '//' axis XML queries efficiently. Among relational approaches, the RP approach with the BLAS extension has the best performance for querying schemaless XML data.

The native approach develops native algorithms to efficiently process θ-joins involved in XML twig queries that are essentially structural joins of inverted lists. In this approach, many existing important components in RDBMS, such as storage management, access methods, query processing and optimization, concurrency control and recovery, have to be rebuilt from scratch. Among native approaches, the Holistic approach shows the best query processing performance in experiments.

Just as [ZND$^+$01] implies, a good approach should be integrating native θ-join algorithms for XML twig queries into existing relational query optimizers so that extended relational query optimizers will be able to process XML twig queries more efficiently. Meanwhile, in this integration approach, other existing important components in RDBMS than query optimizers, such as concurrency control and recovery, can also be fully reused so that development efforts will be significantly saved. Therefore, this integration approach will gain the best trade-off between XML query processing performance and development efforts.

References

[ABJ89] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Management of transitive relationships in large data and knowledge bases. *SIGMOD Conference*, 1989.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. *VLDB Conference*, 2000.

[AKJK+02] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jifnesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: a primitive for efficient XML query pattern matching. *ICDE Conference*, 2002.

[BGKS03] Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation- vs. index-based XML multi-query processing. *ICDE Conference*, 2003.

[BKS02] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. *SIGMOD Conference*, 2002.

[CDZ04] Yi Chen, Susan B. Davidson, and Yifeng Zheng. BLAS: An efficient XPath processing system. *SIGMOD Conference*, 2004.

[Cha02] D. Chamberlin. XQuery: an XML query language. *41 (4)*, 2002.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-Hop labels. *SIAM Journal on Computing*, 32:1338–1355, 2003.

[CVZ+02] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. *VLDB Conference*, 2002.

[DAF+03] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems (TODS)*, 28:467–516, 2003.

[DF03] Yanlei Diao and Michael J. Franklin. High-performance XML filtering: an overview of YFilter. *IEEE Data Engineering Bulletin*, 26:41–48, 2003.

[DFFT02] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and scalable filtering of XML documents. *ICDE Conference*, 2002.

[Die82] Paul F. Dietz. Maintaining order in a linked list. *ACM Symposium on Theory of Computing*, 1982.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22:27–34, 1999.

[Gro04a] W3C Group. Extensible Markup Language (XML). *http://www.w3.org/XML/*, 2004.

[Gro04b] W3C Group. Guide to the W3C XML specification (XMLspec) DTD, version 2.1. *http://www.w3.org/XML/1998/06/xmlspec-report.htm*, 2004.

[Gro04c] W3C Group. XML path language (XPath) 2.0. *http://www.w3.org/TR/xpath20/*, 2004.

[Gro04d] W3C Group. XML Schema. *http://www.w3.org/XML/Schema*, 2004.

[Gro04e] W3C Group. XQuery 1.0: an XML query language. *http://www.w3.org/TR/xquery/*, 2004.

[Gru02] Torsten Grust. Accelerating XPath location steps. *SIGMOD Conference*, 2002.

[GvKT04] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29:91–131, 2004.

[HBG+03] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis Viglas, Yuan Wang, Jeffrey F. Naughton, and David J. DeWitt. Mixed mode XML query processing. *VLDB Conference*, 2003.

[JAKC+02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. TIMBER: A native XML database. *VLDB Journal*, 11:274–291, 2002.

[JLW04] Haifeng Jiang, Hongjun Lu, and Wei Wang. Efficient processing of twig queries with OR-predicates. *SIGMOD Conference*, 2004.

[JWLY03] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. *VLDB Conference*, 2003.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. *VLDB Conference*, 1999.

[OCL04] OCLC. Dewey decimal classification. *http://www.oclc.org/dewey/*, 2004.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-friendly XML node labels. *SIGMOD Conference*, 2004.

[Org04] SAX Project Organizatiion. SAX: Simple API for XML. *http://www.saxproject.org/*, 2004.

[PAKJ+02] Stelios Paparizos, Shurug Al-Khalifa, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Divesh Srivastava, and Yuqing Wu. Grouping in XML. *EDBT Workshops*, 2002.

[PCS+04] Shankar Pal, Istvan Cseri, Gideon Schaller, Oliver Seeliger, Leo Giakoumakis, and Vasili Vasili Zolotov. Indexing XML data stored in a relational database. *VLDB Conference*, 2004.

[PWLJ04] Stelios Paparizos, Yuqing Wu, Laks V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. *SIGMOD Conference*, 2004.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Conference*, 2000.

[SM83] G. Salton and M. J. McGill. Introduction to modern information retrieval. *McGraw-Hill*, 1983.

[SSK+01] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30:20–26, 2001.

[STW04] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An efficient connection index for complex XML document collections. *EDBT Conference*, 2004.

[STW05] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. *ICDE Conference*, 2005.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. *VLDB Conference*, 1999.

[TDCZ02] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record*, 31 (1):5–10, 2002.

[TRP+04] Feng Tian, Berthold Reinwald, Hamid Pirahesh, Tobias Mayr, and Jussi Myllymaki. Implementing a scalable XML publish/subscribe system using a relational database system. *SIGMOD Conference*, 2004.

[TVB+02] Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. *SIGMOD Conference*, 2002.

[WPJ03] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. *ICDE Conference*, 2003.

[YASU01] Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)*, 1:110–141, 2001.

[ZND+01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. *SIGMOD Conference*, 2001.