

INDEXING AND PATH QUERY PROCESSING FOR XML
DATA

by
Quanzhong Li

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2004

UMI Number: 3158121

Copyright 2005 by
Li, Quanzhong

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3158121

Copyright 2005 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

The University of Arizona ®
Graduate College

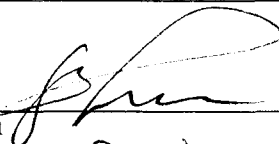
As members of the Final Examination Committee, we certify that we have read the

dissertation prepared by Quanzhong Li

entitled Indexing and Path Query Processing for XML Data

and recommend that it be accepted as fulfilling the dissertation requirement for the

Degree of Doctor of Philosophy



Bongki Moon

8/6/04


date



Richard T. Snodgrass

8/9/04

date



Gregory R. Andrews

8/9/04

date

date

date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

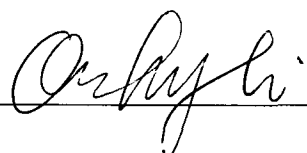


Dissertation Director: Bongki Moon 8/29/04
date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:  _____

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Bongki Moon, for his support, guidance and encouragement throughout my graduate studies.

I would also like to thank Dr. Gregory R. Andrews and Dr. Richard T. Snodgrass for their kind acceptance to serve as the members of my committee, and for spending time to review my dissertation and to provide precious comments.

My friends and colleagues at the Department of Computer Science have always offered an inspiring and friendly atmosphere. In particular, I had invaluable and stimulating discussions with Ines Fernando Vega Lopez and Praveen R. Rao through the course of this work.

To all these persons, I would like to express my deep appreciation.

To my parents, Pigong and Delan.
To my wife, Dengfeng, and my son, Bill.

TABLE OF CONTENTS

LIST OF FIGURES	9
LIST OF TABLES	11
ABSTRACT	12
CHAPTER 1. INTRODUCTION	14
1.1. Background	14
1.2. Research Problems	16
1.3. Approaches and Contributions of the Dissertation	18
1.4. Outline of the Dissertation	21
CHAPTER 2. XML NUMBERING SCHEMES	23
2.1. Introduction	23
2.1.1. The Benefits of Numbering Schemes	23
2.1.2. Preorder and Postorder Traversal Numbering Scheme	24
2.2. Extended Preorder Numbering Scheme	24
2.2.1. Definition	24
2.2.2. Usage	26
2.2.3. Range Mapping and Containment Property	27
2.3. Related Work	29
2.4. Summary	31
CHAPTER 3. XML INDEXING AND STORAGE SYSTEM	32
3.1. Introduction	32
3.2. Index Structure	33
3.3. An Example	35
3.4. Summary	37
CHAPTER 4. ANCESTOR-DESCENDANT PATH QUERY PROCESSING	39
4.1. Introduction	39
4.2. Related Work	41
4.3. Sort-merge Based Algorithms	43
4.3.1. \mathcal{EA} -Join	43
4.3.2. \mathcal{EE} -Join	46
4.3.3. Performance Study of Sort-merge Based Algorithms	48
4.4. Partition-Based Algorithms	60
4.4.1. Data Partitioning	61
4.4.2. Descendant Partition Join	62

TABLE OF CONTENTS—*Continued*

4.4.3.	Segment Tree Partition Join	65
4.4.4.	Ancestor Link Partition Join	66
4.4.5.	Performance Study of Partition-Based Algorithms	68
4.5.	Summary	72
CHAPTER 5.	XML TWIG QUERY PROCESSING	74
5.1.	Introduction	74
5.2.	Related Work	77
5.2.1.	XML Indexing Techniques	77
5.2.2.	XML Path Processing Algorithms	80
5.3.	The CB-tree Index	81
5.3.1.	Motivation	81
5.3.2.	Containment and Reverse Containment Queries	82
5.3.3.	The Containment B ⁺ -tree (CB-tree) Index	84
5.3.4.	Storing Extra Entries in the CB-tree	85
5.3.5.	Operations on the CB-tree	87
5.4.	The IndexTwig Algorithm	88
5.4.1.	The Structure of Twig Query Nodes	89
5.4.2.	Basic Idea of the Algorithm	89
5.4.3.	Outputting Answers	92
5.4.4.	The Range Cache of a Query Node	93
5.4.5.	Output Model	94
5.4.6.	Fast Existence Test	94
5.4.7.	Determining Jump Positions for Child Query Nodes	95
5.4.8.	The IndexTwig Algorithm	96
5.4.9.	Index Independency	100
5.4.10.	Algorithm Illustration	100
5.5.	Performance Study	101
5.5.1.	Data Sets and Performance Metrics	104
5.5.2.	Twig Queries for Performance Study	104
5.5.3.	Index Replication Cost	106
5.5.4.	Algorithm Performance	108
5.6.	Summary	111
CHAPTER 6.	XML PATH PREDICATE PROCESSING	113
6.1.	Introduction	113
6.2.	Related Work	115
6.3.	Predicate Processing Using B ⁺ -trees	117
6.3.1.	Indexing Element	117
6.3.2.	Indexing Value	118

TABLE OF CONTENTS—*Continued*

6.4. The EVR-tree Index	120
6.4.1. Index Structure	120
6.4.2. Advantages of the EVR-tree	120
6.5. Performance Study	126
6.5.1. Experimental Settings	126
6.5.2. Performance Analysis	126
6.6. Summary	130
CHAPTER 7. XML PATH PROCESSING USING RDBMS	131
7.1. Introduction	131
7.2. System Description	133
7.2.1. Mapping XML Data to Relational Schemas	133
7.2.2. XPath Query Engine	137
7.2.3. Web-Based User Interface	139
7.2.4. Implementation	140
7.3. Related Work	142
7.4. Summary	143
CHAPTER 8. CONCLUSIONS AND FUTURE WORK	144
REFERENCES	148

LIST OF FIGURES

FIGURE 1.1.	Sample XML Data, <code>books.xml</code>	15
FIGURE 1.2.	Sample DTD, <code>books.dtd</code>	16
FIGURE 1.3.	Sample XQuery	18
FIGURE 2.1.	Preorder and Postorder Numbering Scheme	25
FIGURE 2.2.	Extended Preorder Numbering Scheme	26
FIGURE 2.3.	Number-Range Mapping	27
FIGURE 2.4.	Range Point Set Example	28
FIGURE 3.1.	Index Structure Overview	33
FIGURE 3.2.	Element Index	34
FIGURE 3.3.	Structure Index	35
FIGURE 3.4.	Book List XML Tree	36
FIGURE 3.5.	<code>book</code> Element List in XISS Element Index	36
FIGURE 3.6.	Sample Structure Index	38
FIGURE 4.1.	Examples of Different Attribute Orders	45
FIGURE 4.2.	An Extreme Case of Element-Element Join	47
FIGURE 4.3.	Total Elapsed Time of Query $E_A//E_B$	53
FIGURE 4.4.	IO Time of Query $E_A//E_B$	54
FIGURE 4.5.	Speed-up of $\mathcal{E}\mathcal{E}$ -Join over Bottom-up	56
FIGURE 4.6.	Total Elapsed Time of Query $E[@A]$	57
FIGURE 4.7.	IO Time of Query $E[@A]$	58
FIGURE 4.8.	Speed-up of $\mathcal{E}\mathcal{A}$ -Join over Top-down and Bottom-up	59
FIGURE 4.9.	Scalability Test of $\mathcal{E}\mathcal{E}$ -Join and $\mathcal{E}\mathcal{A}$ -Join	60
FIGURE 4.10.	End Point Array and Its Segment Tree	66
FIGURE 4.11.	Shakespeare Data Queries	70
FIGURE 4.12.	DBLP Data Queries	70
FIGURE 5.1.	The Tree Pattern of Q2	75
FIGURE 5.2.	Closed Method to Store Extra Entries	86
FIGURE 5.3.	Hybrid Method to Store Extra Entries	86
FIGURE 5.4.	Parent-Child Data Skips	90
FIGURE 5.5.	Parent-Children Data Skips	91
FIGURE 5.6.	A Complex Twig and Its Simple Twigs	92
FIGURE 5.7.	Algorithm Illustration	102
FIGURE 5.8.	Elapsed Time IndexTwig vs. TSGeneric+	109
FIGURE 5.9.	I/O Performance IndexTwig vs. TSGeneric+	110
FIGURE 5.10.	IndexTwig CB-tree vs. XR-tree	110
FIGURE 6.1.	The EVR-tree Illustration	121

LIST OF FIGURES—*Continued*

FIGURE 6.2.	Skip Values in the EVR-tree	122
FIGURE 6.3.	Skip Elements in the EVR-tree	124
FIGURE 6.4.	DBLP Predicate Query Performance	127
FIGURE 6.5.	Synthetic Data Predicate Query Performance	129
FIGURE 7.1.	Tables in Schema A (Primary keys in bold)	135
FIGURE 7.2.	Tables in Schema B (Primary keys in bold)	136
FIGURE 7.3.	XISS/R System Architecture	137
FIGURE 7.4.	XISS Query Interface Example	140

LIST OF TABLES

TABLE 3.1.	Sample <i>nid</i> Assignment	37
TABLE 3.2.	Sample Value Table	37
TABLE 4.1.	XML Data Sets for \mathcal{EA} -Join and \mathcal{EE} -Join	49
TABLE 4.2.	Size of Indexes for Two Different Page Sizes	51
TABLE 4.3.	Summary of \mathcal{EE} -Join Queries	53
TABLE 4.4.	Summary of \mathcal{EE} -Join Queries of Different Path Lengths	55
TABLE 4.5.	Summary of \mathcal{EA} -Join Queries	56
TABLE 4.6.	Queries for Performance Study	69
TABLE 5.1.	The Structure of Query Nodes	88
TABLE 5.2.	Number of I/O Operations: TwigStack, TwigStackXB and Index-Twig	103
TABLE 5.3.	Twig Queries for Performance Study	105
TABLE 5.4.	Result Sizes of Twig Queries	105
TABLE 5.5.	Index Information of the CB-tree and the XR-tree	107

ABSTRACT

XML has emerged as a new standard for information representation and exchange on the Internet. To efficiently process XML data, we propose the extended preorder numbering scheme, which determines the ancestor-descendant relationship between nodes in the hierarchy of XML data in constant time, and adapts to the dynamics of XML data by allocating extra space.

Based on this numbering scheme, we propose sort-merge based algorithms, **\mathcal{EA} -Join** and **\mathcal{EE} -Join**, to process ancestor-descendant path expressions. The experimental results showed an order of magnitude performance improvement over conventional methods. We further propose the partition-based algorithms, which can be chosen by a query optimizer according to the characteristics of the input data.

For complex path expressions with branches, we propose the Containment B^+ -tree (CB-tree) index and the IndexTwig algorithm. The CB-tree, which is an extension of the B^+ -tree, supports both the *containment query* and the *reverse containment query*. It is an effective indexing scheme for XML documents with or without a small number of recursions. The proposed *IndexTwig* algorithm works with any index supporting containment and reverse containment queries, such as the CB-tree. We also introduce a simplified output model, which outputs only the necessary result of a path expression. The output model enables the *Fast Existence Test* (FET) optimization to skip unnecessary data and avoid generating unwanted results.

Also in this dissertation, we introduce techniques to process the predicates in XML path expressions using the EVR-tree. The EVR-tree combines the advantages of indexing on values or elements individually using B^+ -trees. It utilizes the high value selectivity and/or high structural selectivity, and provides ordered element access by using a priority queue.

At the end of the dissertation, we introduce the XISS/R system, which is an im-

plementation of the XML Indexing and Storage System (XISS) on top of a relational database. The XISS/R includes a web-based user interface and a XPath query engine to translate XPath queries into efficient SQL statements.

CHAPTER 1

INTRODUCTION

1.1 Background

The eXtensible Markup Language (XML) has recently emerged as a new standard for information representation and exchange on the Internet [9]. Since XML data is self-descriptive, XML is considered one of the most promising means to define semi-structured data, which is expected to be ubiquitous in large volumes from diverse data sources and applications on the web [2]. XML is extensible. It allows users to make up any new tags for descriptive markup for their own applications. Such user-defined tags on data elements can encode the semantics of data. The relationships between elements can be defined by nested structures and references. As an example, Figure 1.1 shows the content of a book list in XML format. Two `book` elements are contained inside a `books` element. Each book has a `title`, `price`, and one or more `chapter` elements. Each `chapter` may contain a `title` and one or more `section` elements. Inside a `section`, there are `title`, `figure`, `table`, and even `section` elements. Note that a `section` enclosed inside a `section` makes it a recursive nesting. A `section` also has an attribute `sid`. We can see from this example that the flexibility of XML enable us to describe hierarchical data freely. However, this freedom comes at the cost of increased complexity in storing and querying XML data. In this dissertation, we will present techniques to addresses this increased complexity, and make the XML data management easier.

The `DOCTYPE` element on the second line in Figure 1.1 declares the document type definition (DTD) of the XML data. This DTD is shown in Figure 1.2, and it defines the structure of the sample book list data. Applications can use DTD or XML Schema

```
<?xml version="1.0"?>
<!DOCTYPE books SYSTEM "books.dtd">
<books>
  <book>
    <title>Example Book in XML</title>
    <price>59.99</price>
    <chapter>
      <title>Chapter 1</title>
      <section sid="1">
        <title>Section 1.1</title>
        <table caption="Table 1"/>
        <section sid="2">
          <title>Section 1.1.1</title>
          <figure caption="Figure 1"/>
        </section>
      </section>
    </chapter>
    <chapter>
      <title>Chapter 2</title>
      <section sid="3">
        <title>Section 2.1</title>
        <figure caption="Figure 2"/>
        <table caption="Table 2"/>
      </section>
    </chapter>
  </book>
  <book>
    <title>Expensive Book</title>
    <price>119.99</price>
    <chapter>
      <title>Chapter 1</title>
    </chapter>
  </book>
</books>
```

FIGURE 1.1. Sample XML Data, books.xml

```

<!ELEMENT books (book+)>
<!ELEMENT book (title, price, chapter*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (figure | table | section)* )>
<!ATTLIST section
  sid CDATA #REQUIRED>
<!ATTLIST table
  caption CDATA #REQUIRED>
<!ATTLIST figure
  caption CDATA #REQUIRED>

```

FIGURE 1.2. Sample DTD, books.dtd

to define the structure, content and semantics of XML data, which is an important feature of XML. At the same time, constraints on XML data can be expressed using DTD or XML Schema. Then XML documents can be validated according to DTD or XML Schema definitions.

1.2 Research Problems

As more and more information is stored and exchanged in XML, or presented as XML through various interfaces, the ability to intelligently and efficiently query XML data becomes increasingly important. Several query languages have been proposed. Examples are XML-QL [20], XML-GL [12], Quilt [13], XPath [16], and XQuery [7]. The common feature of these languages is the use of path expressions to navigate through arbitrarily long paths and retrieve data from the hierarchy of XML data. XQuery is the public working draft of a query language for XML released from the World Wide Web Consortium (W3C). The XQuery language is designed to be broadly applicable across all types of XML data sources from documents to databases and object repositories. In XQuery, path expressions are used to locate nodes in XML data.

XQuery's path expressions are derived from XPath 1.0 and are identical to the path expressions of XPath 2.0. A path expression consists of a series of one or more steps, separated by a single or double slash. Each step evaluates to a sequence of nodes. Let us use the following path expression as an example.

```
/books/book/chapter
```

The first step, `/books`, selects the `books` element at the root of the XML data. Then it uses the step, `/book`, to get all the `book` elements, and finally uses the step, `/chapter`, to return a sequence of `chapter` elements. The whole expression finds all chapters in the book list. If we apply the above path expression on the sample data in Figure 1.1, we get a sequence of `chapter` elements, two in the first book and one in the second book. The same set of chapters can be found by the following expression, which uses the double slash.

```
//book/chapter
```

The step with a double slash, `//book`, selects all the `book` elements regardless of the level in the document tree. If all `book` elements appear as the children of the element `books`, these above two expressions will return the same result.

In XQuery, predicates can be used to select a subset of the nodes. Each predicate is enclosed in a pair of square brackets. For example, the following example has a predicate to select the `book` elements with a price less than 100.

```
//book[price < 100]
```

The expression inside a predicate does not have to return a boolean value. If an expression results in a non-empty sequence of element nodes, the predicate has an effective boolean true value. A detailed introduction of predicates will be presented in Chapter 6. With the introduction of predicates, a path expression is not just a single path. It can be very complicated. For example, the following path expression has two predicates to filter `section` elements.

```

for    $b in /books/book
where  $b/price > 100 and
       $b/title = "Expensive Book"
return $b

```

FIGURE 1.3. Sample XQuery

Q1: `//book//section[figure][table]/title`

This path expression finds all titles of sections that directly contain both figure and table elements. Applying this query on the sample book list data in Figure 1.1, we get a title element, `<title>Section 2.1</title>`. Since we will use this path expression in the rest of the dissertation frequently, we refer to it as Q1.

Path expressions can be used standalone in XQuery. They can also be combined inside other expressions to construct more complex expressions. We use a simple FLWOR¹ expression as an example to demonstrate how path expressions are used. The XQuery in Figure 1.3 finds the books with the name, `Expensive Book`, and the price greater than 100. In Figure 1.3, the `for` clause iterates through the `book` element sequence and assigns each `book` element to variable `$b`. Each `book` element is then filtered by the `where` clause. Only the qualified `book` element is passed to the `return` clause, which returns the book element. In this XQuery, there are several path expressions, e.g., `/books/book`, `$b/title` and `$b/price`. During the query processing, these path expressions will be evaluated. Doing this efficiently is clearly essential in XML query processing.

1.3 Approaches and Contributions of the Dissertation

In this dissertation, we present the techniques of efficient evaluation of XML path expressions. We take a step-by-step approach. First, simple ancestor-descendant path expressions are addressed. Then we present the techniques to deal with arbitrarily

¹An acronym standing for the first letter of the clauses, `for`, `let`, `where`, `order by`, and `return`.

complex tree patterns. Following this, we describe how to efficiently process path expressions with predicates.

XML data can be modeled by a tree structure, where nodes represent elements, attributes and text data, and parent-child node pairs represent nesting between XML data components. To speed up the processing of path expression queries, it is important to be able to quickly determine ancestor-descendant relationship between any pair of nodes in the hierarchy of XML data. We propose the *extended preorder numbering scheme* to capture the tree structure of XML data. The proposed numbering scheme associates an extended preorder number with elements and attributes, and quickly determines the ancestor-descendant relationship between elements and/or attributes in the hierarchy of XML data.

Using the numbering scheme, sort-merge based algorithms can be used to process parent-child and ancestor-descendant type structure joins. We introduce the **\mathcal{EE} -Join** algorithm in this category for processing path expression queries. In particular, the **\mathcal{EE} -Join** algorithm is highly effective for searching paths that are very long or whose lengths are unknown.

We also discuss the use of partition-based algorithms to process XML join queries. We first formulate XML path queries as range-point join queries. Then we discuss the partition-based algorithms that can utilize the *range containment property* to efficiently process the range-point join queries. Under the partition-based framework, we propose three algorithms, namely *Descendant partition join*, *Segment-tree partition join* and *Ancestor Link partition join*, which can be chosen by a query optimizer by considering different characteristics of input data.

Since indexes can be used to speed up data access and address only relevant data, we also present the techniques to efficiently evaluate XML path expressions based on indexes. We propose the Containment B⁺-tree (CB-tree) index (an extension of the B⁺-tree) that supports both the *containment query* and the *reverse containment query*. When there are no recursively nested elements (in which an element can be

a sub-element of itself) in the XML data, the CB-tree is the same as the B⁺-tree. Based on the CB-tree, we propose a twig join algorithm named *IndexTwig*. The IndexTwig algorithm works with any index supporting containment and reverse containment queries. A simplified output model is proposed and used in the IndexTwig algorithm. Based on this, the *Fast Existence Test* (FET) optimization is used to skip non-matching data and avoid generating unwanted results.

To efficiently process the predicates in XML path expressions, we propose the EVR-tree to index elements and values together. The EVR-tree takes the advantages of indexing on values or elements individually using B⁺-trees. It utilizes the high value and/or structural selectivities, and provided ordered element access by using a priority queue.

Since relational databases are mature and widely used, we propose our implementation of the XML Indexing and Storage System, called *XISS/R*, which is based on relational databases. The XISS/R system demonstrates an efficient approach for using relational database systems to store and evaluate queries on XML data.

The main contributions of the dissertation are:

- The proposed numbering scheme is designed based on the notion of *extended preorder* to accommodate future insertions gracefully. This numbering scheme allows us to determine the ancestor-descendant relationship between elements and attributes in constant time.
- The proposed sort-merge based join algorithm can process path expression queries without traversing the hierarchy of XML data. Experimental results from our prototype system implementation show that the proposed algorithm can process XML queries up to 10 times faster than conventional approaches.
- Under the partition-based framework, the proposed algorithms can be chosen by query optimizer according to different characteristics of the input data.

- The proposed *Containment B⁺-tree (CB-tree)* index, which is a variant of the traditional B⁺-tree, supports both containment and reverse containment queries. Since the CB-tree can be implemented with little modification to the traditional B⁺-tree, existing techniques for the B⁺-tree can be incorporated easily.
- The index based twig join algorithm, *IndexTwig*, can skip unnecessary data and avoid generating intermediate results. The *IndexTwig* algorithm can work with any index supporting containment and reverse containment queries.
- The proposed simple output model and the *Output Optimization (FET)* can be easily combined with the index based twig join algorithm to avoid accessing data needed for generating unwanted results.
- Techniques to index and evaluate path expressions with predicates are proposed.
- We introduce the *XISS/R* system, which is an implementation of the XML Indexing and Storage System (XISS) on top of a relational database.

1.4 Outline of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 introduces the extended preorder numbering scheme, which is the basis of the techniques for indexing and query processing. Based on the numbering scheme, we describe the XML Indexing and Storage System (XISS) in Chapter 3. We present new algorithms, including sort-merge based algorithms (*EE-Join* and *EA-Join*) and partition-based algorithms, for the efficient evaluation of ancestor-descendant path queries in Chapter 4. Chapter 5 provides a description of the CB-tree index and the *IndexTwig* algorithm to process twig pattern path queries. Previous research work on path query processing is also presented in this chapter. We present the evaluation of path expressions with

predicates especially inequality predicates in Chapter 6, whereas Chapter 7 introduces the XISS/R, the relational database implementation of the XML Indexing and Storage System. Finally, Chapter 8 presents the conclusions of this dissertation.

CHAPTER 2

XML NUMBERING SCHEMES

2.1 Introduction

Since XML is becoming a new standard for information representation and exchange on the Internet, the problem of storing, indexing and querying XML documents poses new challenges to database researchers, and has been among the major issues of database research. XML data can be modeled by a tree structure. Many XML indexing techniques proposed recently are based on tree numbering schemes. By assigning a label (numbers or strings) to each tree node, indexes can be built to efficiently support processing of queries based on ancestor-descendant relationships.

2.1.1 The Benefits of Numbering Schemes

XML data can be queried by a combination of value search and structure search. Search by value can be done by matching such XML values as document names, element names/values, and attribute names/values. Search by structure can be done mostly by examining ancestor-descendant relationships given in path expression queries. To facilitate XML query processing by both value and structure searches, it is crucial to provide mechanisms to quickly determine the ancestor-descendant relationship between XML elements as well as fast access to XML values.

In an XML tree, nodes represent elements, attributes and text data, and parent-child node pairs represent nesting between XML data components. To speed up the processing of path expression queries, it is important to be able to quickly determine ancestor-descendant relationship between any pair of nodes in the hierarchy of XML data. For example, a query with a regular path expression `//book//section` (in

the XPath syntax) is to find all `section` elements that are descendants of any `book` element. Once all `book` elements and `section` elements are found, these two element sets can be *joined* to produce all qualified `book-section` element pairs. This join operation can be carried out without traversing XML data trees, if the ancestor-descendant relationship for a pair of `book` and `section` elements can be determined quickly.

2.1.2 Preorder and Postorder Traversal Numbering Scheme

To the best of our knowledge, Dietz's numbering scheme was the first to use a tree traversal order to determine the ancestor-descendant relationship between any pair of tree nodes [25]. His proposition was: *for two given nodes x and y of a tree T , x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal.*

We can label each node of a tree with a pair of integer numbers according to the preorder and postorder traversal orders. If a node is the n^{th} and m^{th} visited node in the preorder and postorder traversals, respectively, then its two labeled integers are n and m . The ancestor-descendant relationship can be determined by examining these pairs of integers. For example, consider a tree in Figure 2.1 whose nodes are annotated by Dietz's numbering scheme. In the tree, we can tell that node (1,7) is an ancestor of node (4,2), because node (1,7) comes before node (4,2) in the preorder traversal (*i.e.*, $1 < 4$) and after node (4,2) in the postorder traversal (*i.e.*, $7 > 2$).

2.2 Extended Preorder Numbering Scheme

2.2.1 Definition

A benefit from the preorder and postorder numbering scheme is that the ancestor-descendant relationship can be determined in constant time by examining the two

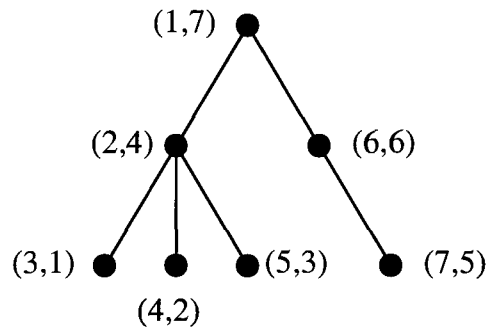


FIGURE 2.1. Preorder and Postorder Numbering Scheme

integers of tree nodes, which are assigned based on the preorder and postorder traversals. On the other hand, the limitation of this approach is the lack of flexibility. Whenever a new node is inserted, the preorder and postorder may need to be recomputed for many tree nodes. To get around this problem, we propose a new numbering scheme called *extended preorder numbering scheme* that associates each node with a pair of numbers $\langle order, size \rangle$ as follows. The *order* is similar to the preorder, and the *size* denotes the range of descendants.

- For a tree node y and its parent x , $order(x) < order(y)$ and $order(y) + size(y) \leq order(x) + size(x)$. In other words, y 's range $[order(y), order(y) + size(y)]$ is contained in x 's range $[order(x), order(x) + size(x)]$.
- For two sibling nodes x and y , if x is the predecessor of y in the preorder traversal, then $order(x) + size(x) < order(y)$. In other words, x 's and y 's ranges are disjoint.

Then, for a tree node x , $size(x) \geq \sum_y size(y)$ for all y 's that are a direct child of x . Thus, $size(x)$ can be an arbitrary integer larger than the total number of the current descendants of x , which allows to accommodate future insertions and deletions gracefully.

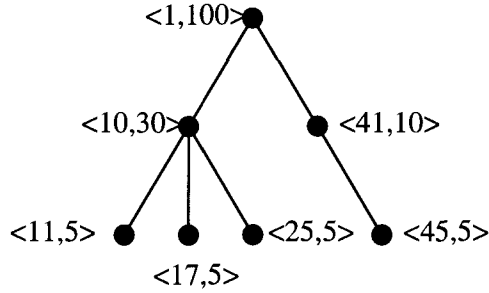


FIGURE 2.2. Extended Preorder Numbering Scheme

2.2.2 Usage

It is not difficult to show that the order of nodes by this proposed numbering scheme is equivalent to that of the preorder traversal. The proposed numbering scheme guarantees that, for a pair of tree nodes x and y , $order(x) < order(y)$ if and only if x comes before y in a preorder traversal. Furthermore, the ancestor-descendant relationship for a pair of nodes can be determined by examining their *order* and *size* values according to the following condition:

For two given nodes x and y of a tree T , x is an ancestor of y if and only if $order(x) < order(y) \leq order(x) + size(x)$.

For example, in Figure 2.2, each node is labeled by a $\langle order, size \rangle$ pair in parentheses according to the extended preorder numbering scheme. In the figure, node (1,100) is an ancestor of node (17, 5), because $17 > 1$ and $17 \leq 100 + 1$. Since we do not require the order to be consecutive integers, there are some gaps in the numbering space, which can be used for future insertions.

Compared with Dietz's scheme, the extended preorder numbering scheme is more flexible and can deal with dynamic updates of XML data more efficiently. Since extra space can be reserved in what we call the extended preorder to accommodate future insertions, global reordering is not necessary until all the reserved space (*i.e.*, unused order values) are consumed. Note that for both numbering schemes, deleting a node

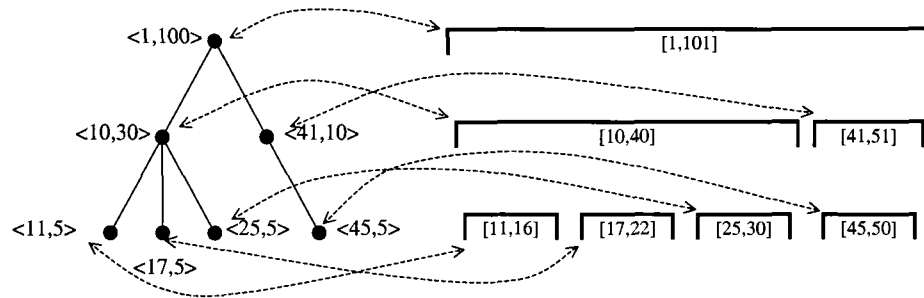


FIGURE 2.3. Number-Range Mapping

does not cause renumbering the nodes. However, it is easier for our numbering scheme to *recycle* the order values of deleted nodes. To do so, we only need to extend the deleting node's siblings' or parent's size value to include the deleted range.

In the rest of the dissertation, both elements and attributes use the *order* of the $\langle order, size \rangle$ pair as their unique identifier, *element ID*, in the document tree. This identifier will be used as the key for sorting and indexing.

2.2.3 Range Mapping and Containment Property

According to the extended preorder numbering scheme, the associated pair of numbers of each node defines a range. The starting point of the range is *order* and the end point is $order + size$. Thus, the XML tree is mapped into a set of ranges, which is illustrated in Figure 2.3. From the right hand side mapped ranges of Figure 2.3, we can easily determine the ancestor-descendant relationship by examining the containment relationship between ranges. For example, since the range $[17, 22]$ is contained in both $[10, 40]$ and $[1, 101]$, the node with order 17 is a descendant of both nodes with order 10 and 1.

Since there is no partial overlap between any two sub-trees, there is no partial overlap among the ranges defined by the numbering scheme. We formalize this property as *range containment property*, which is defined as follows.

Definition 1 (Range Containment Property). *For any two ranges defined by the*

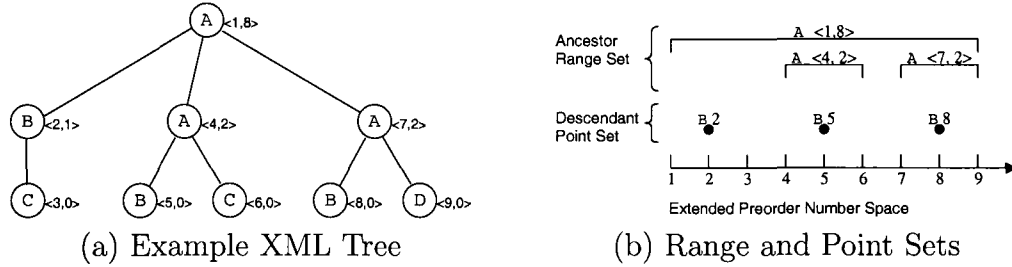


FIGURE 2.4. Range Point Set Example

extended preorder numbering scheme, either one range is contained in the other or they are disjoint.

According to this property, if the starting point of a range is contained in an ancestor range, the whole range is contained in the ancestor range. Thus, for ancestor-descendant type query, we can treat the ancestor set as a range set and the descendant set as a point set. This is illustrated in Figure 2.4. Each node in Figure 2.4(a) has an element name and a pair of numbers next to it.

In Figure 2.4(a), suppose we are going to find all the paths of the pattern $//A//B$, which is to get all the B descendants of A. We can first gather the ancestor set, which is $\{ \langle 1, 8 \rangle, \langle 4, 2 \rangle, \langle 7, 2 \rangle \}$. Then, we obtain the descendant point set, which is $\{ 2, 5, 8 \}$. By the numbering scheme, the ancestor set corresponds to the range set: $\{ [1, 9], [4, 6], [7, 9] \}$. Now, the task to find $//A//B$ is the same as to find the pairs of range and point, where the point is contained in the range. Figure 2.4(b) illustrates this range set and point set relationship. After mapping ancestor sets to range sets and mapping descendant sets to point sets, the problem of finding path patterns is reduced to computing the join between a range set and a point set, which will be referred as *range-point join* in this dissertation. We will present the techniques to process this type of query in the following chapters.

2.3 Related Work

To determine the ancestor-descendant relationships, a document tree can be viewed as a complete k -ary tree with many virtual nodes [46]. The identifier of each node is assigned according to the level-order tree traversal. Then, the ancestors and children of a node can be calculated using just the identifier. The problem of this approach is that when the arity and height of the complete tree are large, the identifier may be a huge number. For example, for a 10-ary complete tree with a height of 10, the total node number will be around 11 billion, which is too large to store in a four-byte word integer. This makes the approach unrealistic for large XML documents.

In Kimber's approach [44], the "tree location address" locates a node in a tree by selecting an ancestor node at each level of the tree. So each identifier of an ancestor node is a prefix of its descendants. Using this prefix-based tree labeling, we need more space to store identifiers, and the time to determine the ancestor-descendant relationship is not constant. It depends on the length of identifiers. For prefix-based tree labeling, techniques to reduce the label size (therefore, to reduce the size of related indexes) has been proposed. Abiteboul *et al.* [3] investigated two static labeling schemes to minimize the label size using a two level partition of an XML tree. Kaplan *et al.* [42] described a prefix-based tree labeling approach, the compressed prefix scheme. This scheme first partitions the tree into paths and then contracts each path into a single virtual node. The original tree labels are obtained using the prefix free binary string assignment of the compressed tree. The ancestor-descendant relationship test is a little more complicated than the prefix test of labels. The compressed prefix scheme can handle the indexing of isolated skewed trees without a large penalty, and produce labels of length matching the upper bound of the simple interval scheme $O(\log N)$, where N is the total number of nodes.

To support XML data which is subject to insertions and deletions, Cohen *et al.* [17] introduced a dynamic labeling scheme to handle the labeling of dynamic in-

sertions of nodes. The proposed scheme reduced labels' size by considering the depth and the width of the XML trees. The authors also considered simple clues such as the size and shape of a subtree to be inserted, and showed the improved upper and lower bounds of the maximum length of the labels by using clues. The Extended Preorder Numbering Scheme is a relatively static numbering scheme. Additional number space can be reserved for insertions, but renumbering cannot be guaranteed. This numbering scheme uses constant time to determine the ancestor-descendant relationship. The prefix numbering scheme is dynamic. The downside is that labels could be large and in different length, which requires more storage space and additional efforts to manage.

Recently, Wang *et al.* proposed a *perfect binary tree* (PBiTree) encoding scheme [64], which can also be used to determine the ancestor-descendant relationship between two elements. In this scheme, a data tree is embedded in a complete binary tree. The in-order tree traversal of the binary tree is used to label the actual data tree nodes. Compared with the Extended Preorder Numbering Scheme, this scheme uses only one number for each node. In the worst case, the number of bits required to represent a code could be large, since there could be a large amount of virtual nodes can not be mapped to real data element nodes. It is also difficult to accommodate insertions in the middle of the tree.

Using the position and depth of a tree node for indexing each occurrence of XML elements has also been proposed [68]. For a non-leaf node, the position is a pair of its beginning and end locations in a depth-first traversal order. The containment properties based on the position and depth are very similar to those of the *extended preorder* independently invented and proposed in this dissertation.

Joe Celko identified several problems of using adjacency list to represent trees in relational databases [11]. For example, the adjacency list model does not model subordination (containment) well. He proposed to use *nested sets* to store a tree in tables. Each node in a tree is assigned two integers, which define a range. And this

labeling scheme is very similar to the extended preorder numbering scheme. These pairs of numbers are stored together with the node in the same tuple. Containment queries such as finding ancestors or descendants can be answered by using these assigned numbers much easier than the adjacency list model.

2.4 Summary

The proposed numbering scheme based on the extended preorder determines the ancestor-descendant relationship between nodes in the hierarchy of XML data in constant time. The numbering scheme can adapt gracefully to the dynamics of XML data objects by allocating a numbering region with extra space.

We also introduced the mapping from tree nodes to ranges and the containment property of these ranges. After the mapping, the problem of finding ancestor-descendant pattern is reduced to range-point join. In the next chapter, we will present the techniques to process this type of join.

CHAPTER 3

XML INDEXING AND STORAGE SYSTEM

As discussed in the previous chapter, the extended preorder numbering scheme provides a way to encode the elements and attributes in an XML document, such that the ancestor-descendant relationship can be determined in constant time, and future insertions can be accommodated gracefully. This numbering scheme also provides opportunities for storing XML data. In this chapter, we describe how XML data are stored inside the XML Indexing and Storage System (XISS). The index structures in the XISS are used by path query processing algorithms that will be discussed in the next chapter. Note that based on the numbering scheme, we can also use relational databases to store and query XML data. This technique is demonstrated in the XML Indexing and Storage System using RDBMS (XISS/R) in Chapter 7.

3.1 Introduction

The XISS provides a framework to utilize the extended preorder numbering scheme. Within the framework, it implements sort-merge based algorithms (\mathcal{EE} -Join and \mathcal{EA} -Join), and partition-based algorithms to process path expressions.

The XISS supports search by element or attribute name and by structure. To achieve this goal, the XISS provides mechanisms to process the following operations efficiently.

- For a given element name string, say **figure**, find a list of elements having the same name (a string *i.e.*, **figure**), grouped by documents they belong to.
- For a given attribute name string, say **caption**, find a list of attributes having

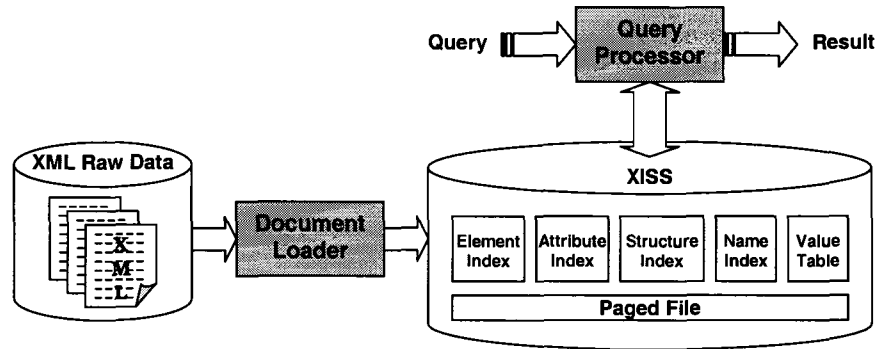


FIGURE 3.1. Index Structure Overview

the same name (a string *i.e.*, *caption*), grouped by documents they belong to.

- For a given element, find its parent element and child elements (or attributes).
For a given attribute, find its parent element.

The XISS is composed of three major components: *element index*, *attribute index* and *structure index*, which are shown in Figure 3.1. The other two components in Figure 3.1 are *name index* for storing name strings and *value table* for attribute and text values.

3.2 Index Structure

Since all value entities in XML data are considered variable-length character strings, all distinct name strings are collected in the *name index*, which is implemented as a B^+ -tree. Then, each distinct name string is uniquely identified by a *name identifier* (or *nid*) returned from the name index. The use of a name index minimizes storage and computational overhead by eliminating replicated strings and string comparisons. For the same reason, all string values (*i.e.*, attribute value and text value) are collected in the *value table*, and each value is assigned a value identifier (*vid*). Each XML document is also assigned a unique document identifier (*did*), which is an index key to retrieve the document name. In the entire system, an element or attribute can be

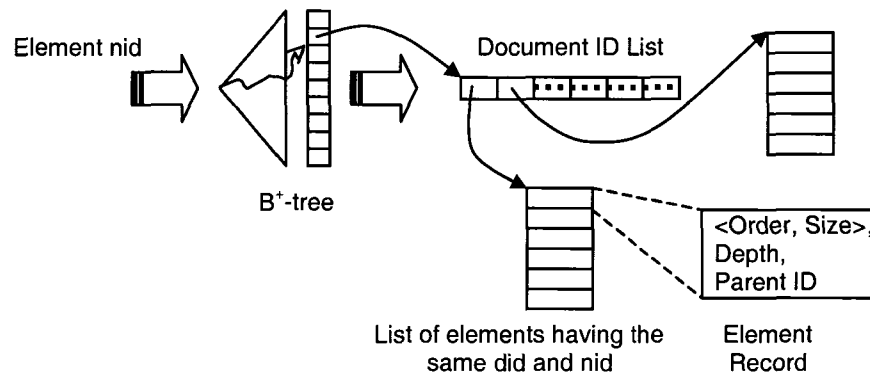


FIGURE 3.2. Element Index

uniquely identified by its *did* and *order* given by the extended preorder numbering scheme.

The *element index*, *attribute index* and *structure index* support the three essential functionalities listed above, respectively. Both the element index and attribute index are implemented as B⁺-trees using name identifiers (*nid*) as keys. Each entry in a leaf node points to a set of fixed-length records for elements or attributes having an identical name string, grouped by documents they belong to. The element index allows us to quickly find all elements with the same tag name. Each element record includes an $\langle order, size \rangle$ pair and other related information of the element. The element records are in a sorted order by the *order* values as shown in Figure 3.2. The attribute index has almost the same structure as the element index, except that the record in attribute index has a value identifier *vid*, which is a key used to obtain the attribute value from the value table.

The organization of the structure index is shown in Figure 3.3. It is a collection of linear arrays, each of which stores a set of fixed-length records for all elements and attributes from an XML document. Within an array, the elements and attributes are together sorted by their *order* value (*i.e.*, in preorder traversal). Each record of the structure index stores a name identifier (*nid*), *order* values of the first sibling, first child, and the first attribute and so on. To retrieve a record, we can use the

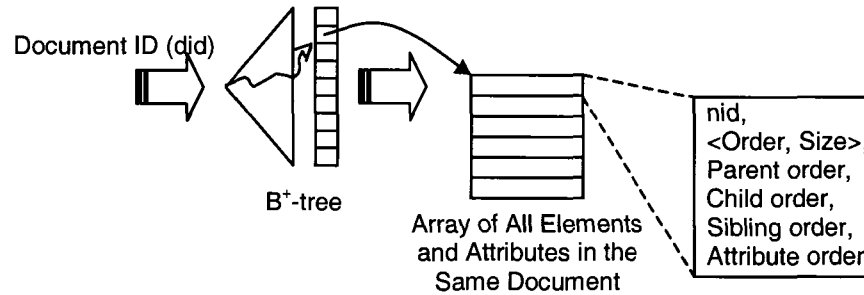


FIGURE 3.3. Structure Index

order value as the array index to quickly locate the record. The structure index stores the information about the tree structure of a document. Using the structure index, an element or attribute record can be found quickly. From the link information inside the record, we can find the parent and children of an element (or attribute). Thus the structure index can be used to support tree traversal operations efficiently. The tree traversal-based algorithms, which will be described in the next chapter, are implemented based on this structure index. Additionally, using the same structural information, XML raw documents can be reconstructed and exported by simulating a preorder traversal using the structure index.

3.3 An Example

We shall use the book list XML data in Figure 1.1 to illustrate how the data are stored in the XISS. Figure 3.4 shows the document tree labeled with two integer numbers according to the extended preorder numbering scheme. Since there is only one document, let us assume the *did* is 1. Table 3.1 shows a sample assignment of element name identifiers (*nid*). The sample value table is shown in Table 3.2. For the element index, let us use the **book** element as an example to show the element index structure. Since we have two **book** elements, there are two records in the **book** element list. The content of these records can be seen in Figure 3.5. The field names inside each record are also displayed in the figure. Since the attribute index has

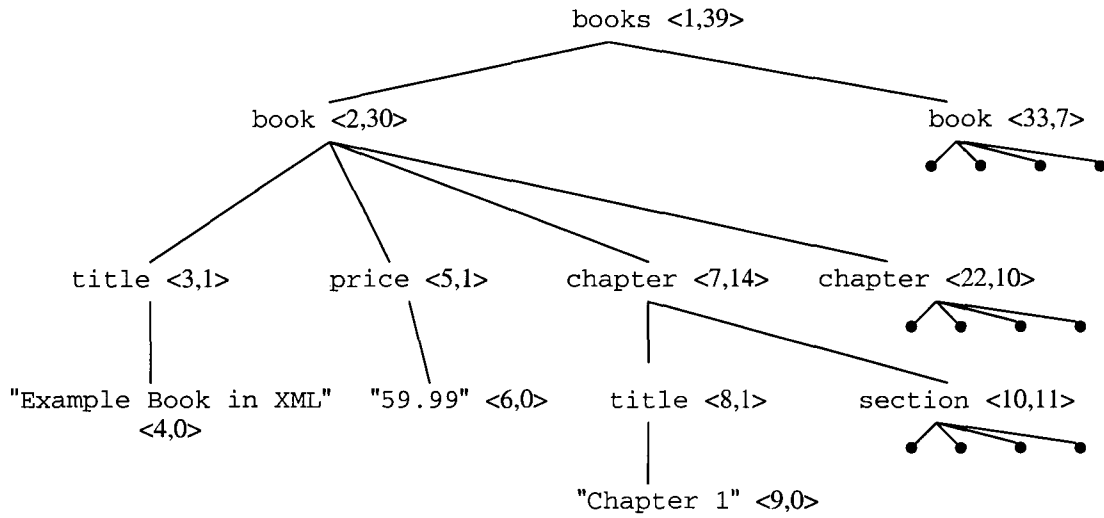


FIGURE 3.4. Book List XML Tree

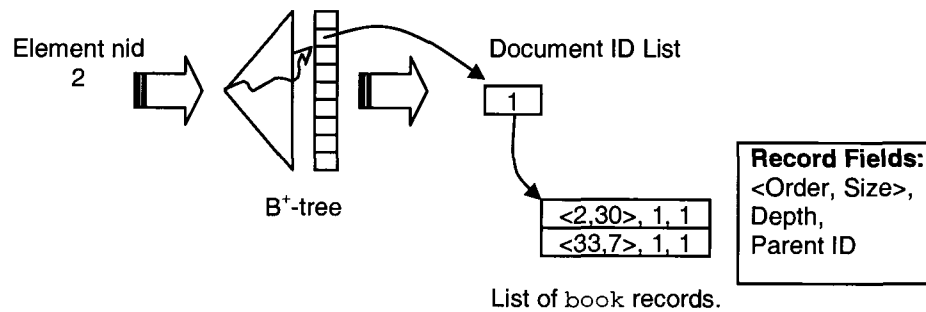


FIGURE 3.5. book Element List in XISS Element Index

similar structures, it is not illustrated in this example.

We also display the structure index of the document in Figure 3.6. The first 10 records are shown in the figure, which is enough to show the details of this structure. Since value records do not have *nid*, the *nid* fields are empty (value 0). The value id *vid* of a value record is stored in the attribute order field. We can see that the records are sorted according to the *order* values. Given an *order* value, we can quickly locate its corresponding record by using the value as the array index.

Element Name	<i>nid</i>
books	1
book	2
title	3
price	4
chapter	5
section	6
table	7
figure	8
sid	9
caption	10

TABLE 3.1. Sample *nid* Assignment

Value String	<i>vid</i>
"Example Book in XML"	1
"59.99"	2
"Chapter 1"	3
"1"	4
"Section 1.1"	5
"Table 1"	6
"2"	7
...	...

TABLE 3.2. Sample Value Table

3.4 Summary

In this chapter, we first described the functionalities that needed to support efficient path processing. Then, we presented the XISS that can be used to store and index XML data. Several index structures of the XISS were introduced, among which element and attribute indexes can be used to retrieve a list of elements or attributes, and structure index can be used to support efficient tree traversals. In the next chapter, we will describe the path expression processing algorithms based on the index structures in the XISS.

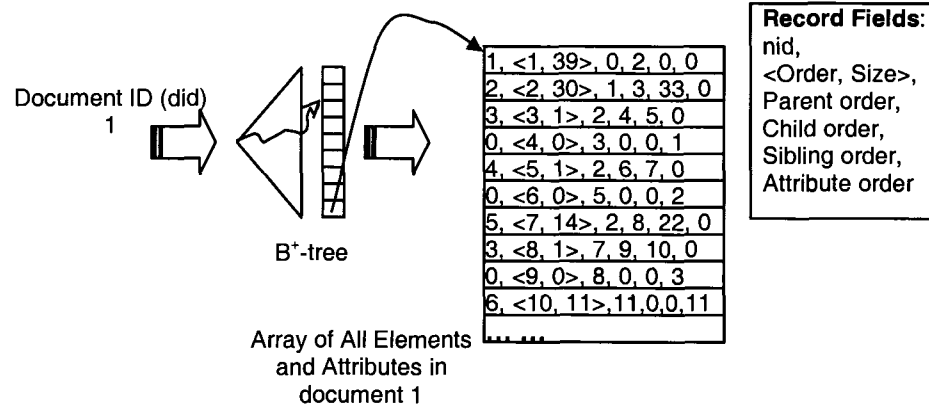


FIGURE 3.6. Sample Structure Index

CHAPTER 4

ANCESTOR-DESCENDANT PATH QUERY PROCESSING

4.1 Introduction

Basic path expressions such as ancestor-descendant and parent-child path queries appear commonly in XML path expressions. For example, consider the following sample query Q1.

```
Q1: //book//section[figure][table]/title
```

The query Q1 is to find all `title` elements that are children of `section` elements, and have `figure` and `table` elements as its siblings in a book. To evaluate this query, we can decompose it into a set of ancestor-descendant and parent-child type of basic path expressions, and join the results of these basic path expressions together in the end. After the decomposition of Q1, we obtain the basic path expressions, `book//section`, `section/figure`, `section/table` and `section/title`. The sub-expression, `book//section`, is an ancestor-descendant type of path expression. This basic path expression is to find all `book` and `section` pairs. The other three are parent-child type of path expressions. From this example, we can see that finding answers of basic path expressions efficiently is clearly important in the whole XML path expression evaluation.

Most straightforward approaches to processing path expression queries, for instance `book//section`, traverse the hierarchy of XML objects in either *top-down* or *bottom-up* fashion [51]. To process the query by a top-down approach, all downward paths starting from a `book` element should be followed to determine whether there exists any `section` element as a descendant. This step needs to be repeated for all

`book` elements in the XML database. This implies that it is absolutely necessary to examine every possible path from each `book` element to all leaf nodes in an XML tree, because it is not usually known where `section` elements will be found in the paths. If the `book` element is the root of an XML tree, then the entire tree will be traversed.

The cost of tree traversal may be reduced by a bottom-up approach. For the same query, `book//section`, all `section` elements will be located. Then, from each of such `section` elements, a corresponding XML tree will be examined by traversing up the tree to find out whether there exists any `book` element as an ancestor. This upward traversal will be simpler and less costly, because there exists always at most one upward path. However, if there are many `section` elements and only a few `book` elements, the cost of bottom-up approach might be even higher than that of top-down approach.

A hybrid approach has been proposed that traverses in both top-down and bottom-up fashions, meeting in the middle of a path expression [51]. This hybrid approach can take advantage of top-down and bottom-up approaches for XML data of certain structural characteristics. However, its effectiveness is not always guaranteed.

With the introduction of XML numbering schemes, new indexing techniques and algorithms can be used to process path queries more efficiently. In the rest of this chapter, we first introduce the related work in Section 4.2. Then we describe the techniques to process ancestor-descendant path expressions. Since the parent-child path expression can be considered as a special case of the ancestor-descendant path (with depth difference limited to one), we will focus on the ancestor-descendant path processing. Two kinds of algorithms, sort-merge based and partition-based algorithms, will be introduced in Section 4.3 and Section 4.4, respectively. Section 4.2 discusses related work about the ancestor-descendant path processing. We summarize this chapter in Section 4.5.

4.2 Related Work

For XML databases with graph-based data models, path traversals play a central role in query processing, and optimizing navigational path expressions is an important issue. The optimal query plan depends not only on the *values* in the database but also on the *shape* of the graph containing the data. Three query evaluation strategies have been proposed for Lore’s cost-based query optimizer [51]. They are a top-down strategy for exploiting the path expression, a bottom-up strategy for exploiting value predicates, and a hybrid strategy. To speed up query processing in a Lore database, four different types of index structures have been proposed [32, 52]. Value index and text index are used to search objects that have specific values; link index and path index provide fast access to parents of an object and all objects reachable via a given labeled path. In the XISS, since we only considered string values, we have only one value table, which serves similar functionalities as the value and text indexes in Lore. The structure index of the XISS provides efficient structural access, including finding parent, which is similar to link index. For the path index, we have to know the path we need to index. However, this information is always known beforehand.

In the *index fabric* [18], all the root-to-leaf element paths in XML data trees are inserted into a disk-resident Patricia trie as strings. The key idea is to transform a memory-based Patricia trie into a block-structure counterpart, so that it can store and access the disk-resident element paths efficiently. Note that the index fabric does not store any suffix of the paths. Thus, a sub-path match operation (a path not fully specified from the root to a leaf) is posed as a query with a wildcard as a prefix and/or a suffix. Although the index fabric can process exact-match path queries efficiently, such queries with wildcards in their paths might be difficult to process with the index fabric. We will show that using the extended preorder numbering scheme, this type of queries can be handled by sort-merge or partition-based algorithms efficiently.

The problem of optimizing regular path expressions has been studied in the context

of navigating semi-structured data in web sites [4, 27]. The semi-structured data is modeled as an edge-labeled graph, where nodes denote HTML pages and edges denote hyperlinks. Abiteboul and Vianu [4] dealt with a path query evaluation that takes advantage of local knowledge (*i.e.*, path constraints) about data graphs that may capture structural information about a web site. They addressed the issue of equivalence decidability of regular path queries under such constraints. Fernández and Dan Suciu [27] proposed two query optimization techniques to rewrite a given regular path expression into another query that reduces the scope of navigation. The above techniques to process path expressions are all based on the graph model. Since XML data can be model by a tree structure, using graph model cannot capture all the features of XML data. Our proposed numbering scheme, on the other hand, uses numbers to keep the structural information of XML data, which enables us to use conventional relational database techniques to process XML data.

Several sort-merge based algorithms have been proposed to process ancestor-descendant type path expressions. Zhang *et al.* proposed to use the position and depth of a tree node for indexing each occurrence of XML elements [68], and proposed a variation of the traditional merge join algorithm, called the multi-predicate merge join (MPMGJN) algorithm, to process containment join (corresponding to ancestor-descendant and parent-child joins). The MPMGJN algorithm showed an order of magnitude performance improvement over standard join algorithms in relational databases. Later, Al-Khalifa *et al.* developed a family of stack-tree structural join algorithms, which utilized in-memory stacks to hold ancestor nodes [61].

Hash-based join algorithms can be used as an alternative to sort-merge based algorithms to process equality joins. Comparisons of sort-based and hash-based algorithms show that many dualities exist between the two types of algorithms and both should be available in a query-processing system [33]. This is also one of the motivations for us to investigate partition-based algorithms, which can be considered as hash-based algorithms. A *partitioned band join* [22] algorithm has been proposed

for evaluation “Band Joins”, which is a class of non-equijoin. There are also a large amount of work has been done in the temporal database area to process temporal intersection joins [35], in which join predicates over time attributes are mostly of the inequality type. To process valid-time joins, a partition-based evaluation algorithm has been proposed [60]. This algorithm utilizes in-memory cache to store “long-lived” tuple and avoids the replication of tuples in multiple partitions.

For XML data, with the introduction of numbering schemes, partition-based algorithms can be used to process the join between range and point sets (containment join). Wang *et al.* [64] proposed a containment query processing framework based on a new coding scheme, PBiTree code, in which a data tree is embedded in a complete binary tree. Partition-based algorithms were also proposed along with the PBiTree encoding. The partitioning strategies used in the paper were limited to the PBiTree encoding. False hits could be introduced by the *roll-up* operation, which makes post-processing to be necessary.

4.3 Sort-merge Based Algorithms

In this section, we describe two sort-merge based algorithms, **\mathcal{EA} -Join** and **\mathcal{EE} -Join**. These two algorithms utilize the XISS index structures introduced in the previous chapter. In Section 4.3.3, comparisons to traversal-based algorithms are studied.

4.3.1 \mathcal{EA} -Join

The **\mathcal{EA} -Join** algorithm joins two intermediate results from subexpressions, which are a list of elements and a list of attributes. The intermediate result can be the result from an index access from the XISS. For example, the path expression, `figure[@caption = "Figure 1"]`, searches all `figure` elements with a caption `Figure 1` from all XML documents in the XISS system. The input to the **\mathcal{EA} -Join** algorithm

Algorithm 1: \mathcal{EA} -Join: Element and Attribute Join

Input: E_1, \dots, E_m : E_1 to E_m are sorted by *did*, each E_i itself is also a list of elements having a common document identifier sorted by *order* values;

A_1, \dots, A_n : A_1 to A_n are sorted by *did*, each A_j itself is also a list of attributes having a common document identifier sorted by *order* values;

Output: A set of (e, a) pairs such that the element e is the parent of the attribute a .

// Sort-merge E_1, \dots, E_m and A_1, \dots, A_n by document identifier, *did*.

```

1: foreach  $E_i$  and  $A_j$  with the same did do
    // Sort-merge list  $E_i$  and list  $A_j$  by order value.
2:   foreach pair  $(e, a)$  during the scan do
        //  $e$  is the element pointed to by the cursor of  $E_i$ 
        //  $a$  is the attribute pointed to by the cursor of  $A_j$ 
        // Test if  $e$  is the parent of  $a$ 
3:     if  $(e.order < a.order \wedge e.order + e.size \geq a.order \wedge$ 
         $e.depth = a.depth - 1)$  then
        output  $(e, a)$ ;
    end
  end
end

```

is a list of `figure` elements and a list of `caption` attributes grouped by documents which they belong to. The \mathcal{EA} -Join algorithm is described in Algorithm 1.

Since the element (or attribute) index maintains the element (or attribute) records in a sorted order by document identifiers and then *order* values, the join of the two lists can be obtained by a *two-stage sort-merge* operation without additional cost of sorting. An element list and an attribute list are merged by document identifiers in the first stage. Then, in the second stage, for a pair of element list and attribute list with a matching document identifier (*i.e.*, extracted from the same document), the elements and attributes are merged by examining the parent-child relationship based on their *order* values given by the numbering scheme.

It is important to ensure that attributes are placed before their sibling elements

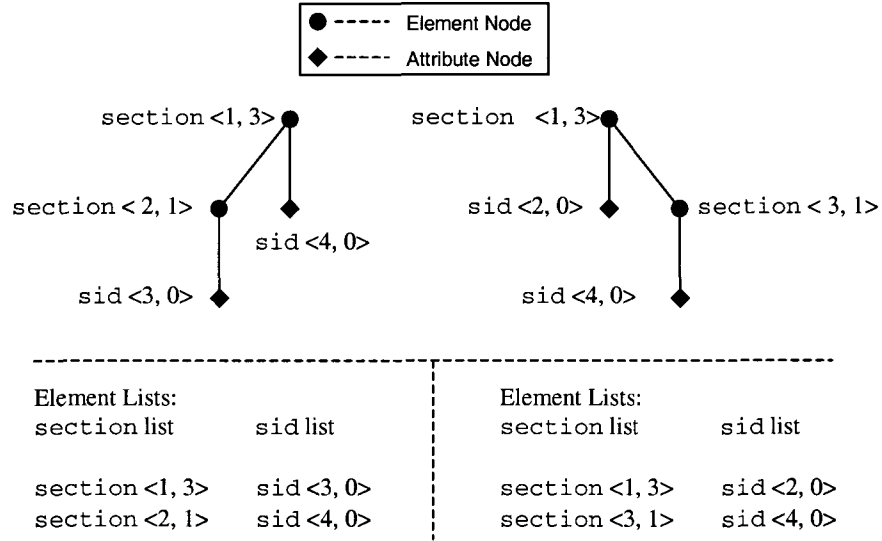


FIGURE 4.1. Examples of Different Attribute Orders

in the order by the numbering scheme. Although we may think it is natural that attributes should be ordered before their element siblings, still we would like to point this out. Its performance impact on the \mathcal{EA} -Join operation is potentially very high, because this additional requirement on the numbering scheme guarantees that those elements and attributes with a matching document identifier can be merged in a *single scan*. Specifically, both the lists $\{E_i\}$ and $\{A_j\}$ grouped by document are scanned once by the outer **foreach** loop (line 1 in Algorithm 1), and both the element list E_i and attribute list A_j are scanned once by the inner **foreach** loop (line 2 in Algorithm 1).

This can be best explained by an example shown in Figure 4.1. Note that tree nodes are annotated by $\langle order, size \rangle$ pairs. The element lists of the two XML trees are shown in the lower part of the figure. Consider the XML tree at the left hand side of Figure 4.1, where an attribute $sid\langle 4, 0 \rangle$ is numbered *after* its sibling element $section\langle 2, 1 \rangle$. By the time the parent-child relationship between $section\langle 1, 3 \rangle$ and $sid\langle 4, 0 \rangle$ is examined, the attribute $sid\langle 3, 0 \rangle$ has already been passed over. Consequently, to examine the parent-child relationship between $section\langle 2, 1 \rangle$ and

Algorithm 2: \mathcal{EE} -Join: Element and Element Join

Input: E_1, \dots, E_m and F_1, \dots, F_n : in sorted order by document identifier
 E_i or F_j is a list of elements having a common document identifier.
Output: A set of (e, f) pairs such that the element e is an ancestor
of the element f .

```

// Sort-merge  $E_1, \dots, E_m$  and  $F_1, \dots, F_n$  by document identifier.
1: foreach  $E_i$  and  $F_j$  with the same did do
    // Sort-merge  $E_i$  and  $F_j$  by order values
2:   foreach pair  $(e, f)$  do
        //  $e$  is the element pointed to by the cursor of  $E_i$ 
        //  $f$  is the element pointed to by the cursor of  $F_j$ 
        // Test if  $e$  is an ancestor of  $f$ 
3:     if  $(e.order < f.order \wedge e.order + e.size \geq f.order)$  then
            output  $(e, f)$ ;
        end
    end
end

```

`sid<3,0>`, the attribute lists must be rescanned. In contrast, in the XML tree at the right hand side, the attribute `sid<2,0>` is numbered *before* its sibling element `section<3,1>`. The parent-child relationships for `section<1,3>` and `sid<2,0>` pair and `section<3,1>` and `sid<4,0>` pair can be determined without rescans.

4.3.2 \mathcal{EE} -Join

The \mathcal{EE} -Join algorithm joins two intermediate results, each of which is a list of elements obtained from a subexpression. For example, a regular path expression `chapter//figure` searches all `chapter-figure` pairs that are in ancestor-descendant relationship from all XML documents. The input to the \mathcal{EE} -Join algorithm is a list of `chapter` elements and a list of `figure` elements grouped by documents which they belong to. The \mathcal{EE} -Join algorithm is described in Algorithm 2.

Like the \mathcal{EA} -Join algorithm, the \mathcal{EE} -Join algorithm can perform the join of two element lists by a *two-stage sort-merge* operation without additional cost of sorting.

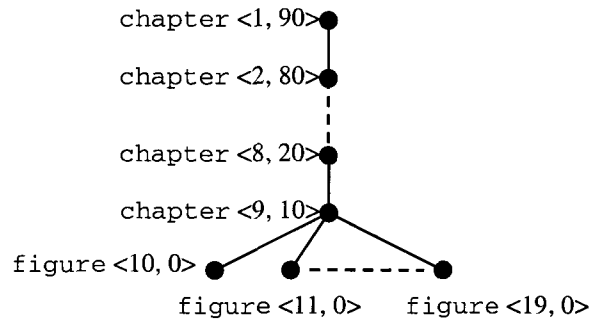


FIGURE 4.2. An Extreme Case of Element-Element Join

Both the element lists are merged by document identifiers in the first stage. Then, in the second stage, for a pair of element sets with a matching document identifier (*i.e.*, extracted from the same document), both the element sets are merged by examining the ancestor-descendant relationship based on their $\langle order, size \rangle$ values given by the numbering scheme.

Unlike the \mathcal{EA} -Join algorithm, however, there is no guarantee that two sets of elements with a matching document identifier can be merged in a single scan by the \mathcal{EE} -Join algorithm. By the extended preorder numbering scheme, for a pair of elements `chapter` and `figure` as an example, their ancestor-descendant relationship is determined by examining whether the $order(\text{figure})$ (*i.e.*, a point in extended-preorder) is contained in $[order(\text{chapter}), order(\text{chapter}) + size(\text{chapter})]$ (*i.e.*, a range in extended-preorder). The join of two sets of elements by ancestor-descendant relationship can be viewed as a join of a range set and a point set. Just as a point can be contained in more than a range, an element `figure` can be a descendant of more than a `chapter` element. See Figure 4.2 for an extreme case, where every `chapter` element must match every `figure` element. Thus, it may be necessary to scan the list of `figure` elements more than once.

Despite the fact that an element set may have to be scanned multiple times by the inner `foreach` loop (line 2 in Algorithm 2), the \mathcal{EE} -Join algorithm is still highly efficient, particularly for searching paths that are long or whose lengths are unknown.

In Section 4.3.3, we compare the \mathcal{EE} -Join algorithm with conventional approaches based on tree traversal. The effectiveness of the \mathcal{EE} -Join is corroborated by the experimental results. Also, in Section 4.2, we will see that related work has been proposed to improve the performance of sort-merge based algorithms by introducing in-memory stacks to cache ancestor elements to avoid re-reads of element lists.

It is worth noting that the \mathcal{EE} -Join algorithm can process an element-element join with a fixed-length path, such as `chapter/title` (a parent-child type join) and `chapter/*/*/figure`, which searches `chapter-figure` element pairs that are in great-grandparent relationship. Using the depth stored with each element in an XML tree, the algorithm can determine the great-grandparent relationship in constant time. With an easy extension, the \mathcal{EE} -Join algorithm can process a subexpression such as `chapter//[@caption]`. Although this subexpression contains a pair of element and attribute, \mathcal{EA} -Join algorithm cannot process it in a single scan. Thus, this subexpression should be processed by \mathcal{EE} -Join algorithm.

4.3.3 Performance Study of Sort-merge Based Algorithms

We implemented the prototype of the XISS to store the XML data and index. A primitive query interface was provided in C++. The Gnome XML parser was used to parse XML data [66]. We also used the GiST C++ library [37] for B⁺-tree indexing. Query processing was directly implemented using the query interface.

Experiments were performed on a Sun Ultra Enterprise 1 Model 170 workstation with UltraSPARC 167MHz CPU running Solaris 2.6. This workstation has 256 MB of memory and a Seagate ST42400N SCSI disk drive (with 5400rpm and 11ms seek time). The disk is locally attached to the workstation and used to store XML data and index. We used the direct I/O feature of Solaris for all experiments to avoid operating system's cache effects.

Data Set	Total Document Size	Documents	Elements	Attributes
Shakespeare	7713KB	37	327K (22)	0 (0)
SIGMOD	3440KB	989	839K (47)	4775 (3)
DBLP	58841KB	783	2666K (29)	199K (3)
NITF100	7531KB	100	63K (124)	263K (142)
NITF1	5194KB	1	38K (86)	171K (106)

TABLE 4.1. XML Data Sets for \mathcal{EA} -Join and \mathcal{EE} -Join

Data Sets and Performance Metrics We have chosen three data sets (Shakespeare, SIGMOD, DBLP) from real-world applications and two synthetic data sets generated by the XML Generator from IBM [24]. These data sets are described in the following, and the characteristics of the data sets are summarized in Table 4.1. In the last two columns, the two numbers in each entry represent the total number of elements (or attributes) and the number of distinct elements (or attributes), respectively.

Shakespeare’s Plays: This data set is the Shakespeare’s plays in XML format, which is marked up by Jon Bosak [19]. It can be obtained from “OASIS the XML Cover Pages” web site [19].

SIGMOD Record: This data set is the XML version of ACM SIGMOD Record [1]. There are many small files, each of which contains an on-line paper or article of the SIGMOD Record.

DBLP: The DBLP is a computer science bibliography [49], which consists of a large number of small XML documents. Each document corresponds to a publication of journals, conferences and books, etc. The current DBLP lists more than 236,000 articles. In our experiment, we used the conference portion of the DBLP. The raw data size is about 58MB.¹

NITF100 and NITF1: These two data sets are generated from the XML Gen-

¹We combined all small files that belong to the same conference into a larger file, which ended up with 783 files, one for each conference.

erator [24], which can randomly generate XML files from a given document type definition (DTD). The DTD we used is the News Industry Text Format (NITF) 2.5 developed by the International Press Telecommunications Council [38]. NITF is an XML-based DTD designed for the markup and delivery of news content. We generated two different XML data sets based on this DTD. One data set was stored in a single large document file (NITF1), and the other was stored in 100 separate document files (NITF100). This can help us to show the effects of different file sizes and file numbers on the XISS system.

Table 4.1 shows that the Shakespeare data set has no attributes and the SIGMOD data set has a very few attributes. For the synthetic data sets NITF100 and NITF1, a large portion of the data objects are attributes. These characteristics can affect the performance of different algorithms. We will show this from the experimental results in the following sections.

In the experiments, we measured the elapsed time (both CPU and IO) of query processing. Since the cost of output generation is the same regardless of algorithms applied, the output cost was not included in the measurements.

Data Loading and Index Construction To load an XML file into the XISS system, the *document loader* (shown in Figure 3.1) first parses the XML data and builds a document tree. While data is being loaded, the document loader assigns an $\langle order, size \rangle$ pair to each element or attribute according to the extended preorder numbering scheme. Then, from this document tree, the document loader builds indexes and inserts all data objects into the XISS system.

The elements (or attributes) having a common name string from the same XML document are grouped together and organized as a sorted list. In the current implementation of the XISS system, each list of elements (or attributes) per each document is stored in a separate group of disk pages. The number of lists may be very large.

Data Set	Page Size 512		Page Size 1024	
	Index Size (KB)	Ratio	Index Size (KB)	Ratio
Shakespeare	21328	2.7	21348	2.7
SIGMOD	15840	4.5	25512	7.2
DBLP	176550	3.0	178890	3.0
NITF100	26212	3.4	33164	4.3
NITF1	13872	2.6	13896	2.6

TABLE 4.2. Size of Indexes for Two Different Page Sizes

Thus, for a large disk page, the storage waste due to internal page fragmentation may not be trivial.

Table 4.2 shows the sizes of the indexes generated by the XISS system for the data sets used in the experiments for two different page sizes. The ratio column is the ratio of the size of index to the size of raw data set (shown in Table 4.1). In the Shakespeare, the DBLP and the NITF1 data sets, individual XML files are large, and an element tends to have many occurrences in XML documents. Thus, the amount of storage waste was not substantial for these data sets, and the sizes of indexes were almost the same for two different page sizes. On the other hand, in the SIGMOD and the NITF100 data sets, there are many small files, and many elements having only a few occurrences in XML documents. Since the effect of internal page fragmentation becomes more pronounced for larger pages, the aggregate sizes of indexes for those data sets increase substantially, when the page size increases. The column *Ratio* in Table 4.2 shows the ratio of the aggregate size of indexes to the size of raw XML data. Throughout the experiments to be described next, we used 1024 bytes as the page size.

Performance of Query Processing In this section, we present the performance measurements and analyze the proposed algorithms mostly for element-element join and element-attribute join operations. The conventional top-down and bottom-up methods are compared with the proposed algorithms. Because the cost of output gener-

ation is the same regardless of algorithms applied, the output cost was not included in the measurements.

Single Element or Attribute Query A path query with only a single element or attribute can be processed by the following steps. First, for a given element or attribute name string, a name identifier (*nid*) is retrieved from the *name index*. Second, using the name identifier, a set of elements (or attributes) grouped by their document identifiers (*did*) are retrieved from the element (or attribute) index.

If those elements (or attributes) are to be used as an intermediate result for the next stage of query processing, they are not actually retrieved from storage until needed by the next operation such as \mathcal{EE} -Join or \mathcal{EA} -Join operations. Instead, a reference to the entry of the element (or attribute) index will be returned to keep the processing cost for this type of query low and almost constant.

Performance of \mathcal{EE} -Join The queries we used for element-element join operations are of the form $E_A//E_B$. For example, the query `chapter//figure` finds all `figure` elements that are descendants of a `chapter` element. The actual queries used in the experiments are shown in Table 4.3. Note that the performance of the DBLP data set is shown in the next paragraph together with the performance of fixed length path tests.

With all these queries, we compared the \mathcal{EE} -Join algorithm with the bottom-up method. The top-down method was not used, because it was expected to be outperformed by the bottom-up method for the data sets. The bottom-up method processes queries in the following steps. First, search all elements with name E_B . Second, starting from each element E_B , traverse up the tree to find E_A elements. Third, if such an element E_A is found, references to the matching E_B elements are returned. As we have described in Chapter 3, the XISS index structure can efficiently support tree traversal operations, which enable us to implement traversal-based algorithms efficiently.

Data Set	Element E_A	Element E_B	# of E_A	# of E_B	# of $E_A//E_B$
Shakespeare	ACT	SPEECH	185	31028	30951
SIGMOD	articles	author	483	9836	7440
NITF-100	body.content	block	3476	3476	4411
NITF-1	body.content	block	1946	2801	5174

TABLE 4.3. Summary of \mathcal{EE} -Join Queries

Consequently, the performance study was fair to tree traversal-based algorithms.

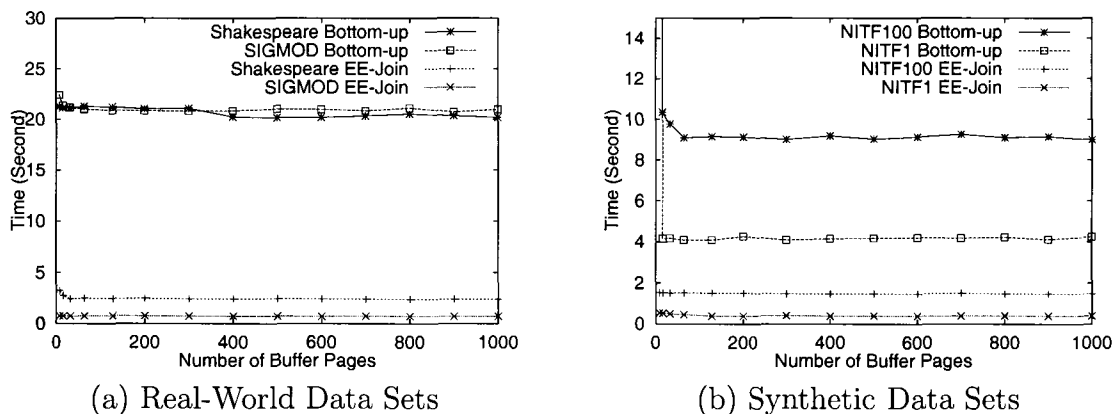
FIGURE 4.3. Total Elapsed Time of Query $E_A//E_B$

Figure 4.3(a) and Figure 4.3(b) show the elapsed time for two real-world data sets (Shakespeare and SIGMOD) and synthetic data sets (NITF100 and NITF1), respectively. The \mathcal{EE} -Join algorithm performs well even for a small number of buffer pages. The bottom-up method takes longer time to process the same query, especially for synthetic data, if the size of buffer pool is small. This is because the \mathcal{EE} -Join algorithm accesses the sorted elements from disk in a sequential manner, while the bottom-up method accesses elements from the structure index almost randomly. This results in a relatively low rate of page faults for \mathcal{EE} -Join algorithm, and a relatively high rate of page faults for the bottom-up method. The vertical lines in Figure 4.3(b) show the severely elongated processing times by the bottom-up method in the extreme case of using only a small number of buffer pages. Obviously, beyond the point where

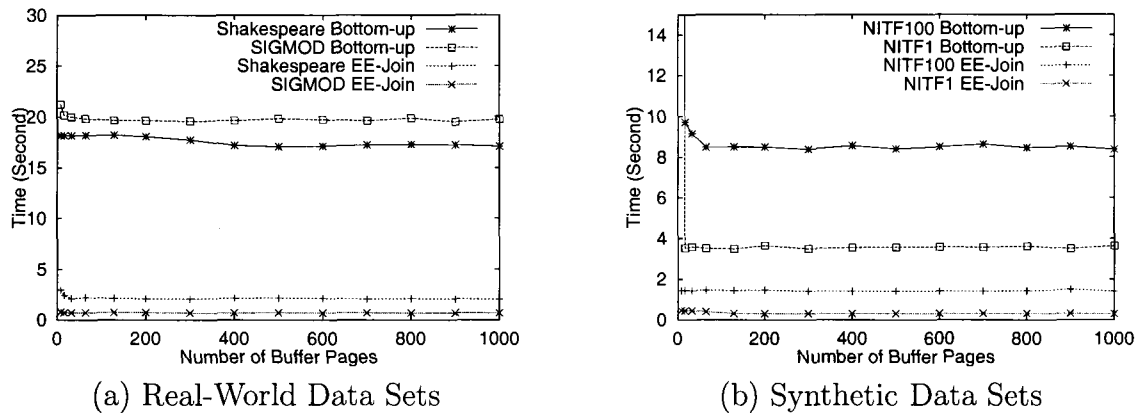


FIGURE 4.4. IO Time of Query $E_A // E_B$

more than enough buffer pages are available, all performance measurements remain constant irrespective of the number of buffer pages.

From all the experiments done with both real-world and synthetic data sets, the \mathcal{EE} -Join algorithm outperformed the bottom-up method by a wide margin. For real-world data sets, the \mathcal{EE} -Join algorithm was *an order of magnitude faster* than the bottom-up method. For synthetic data sets, the \mathcal{EE} -Join algorithm was *about 6 to 10 times faster* than the bottom-up method.

The corresponding IO time measurements are shown in Figure 4.4 for both real-world and synthetic data sets. It shows the same trend in performance as in Figure 4.3. It comes obvious from Figures 4.3 and 4.4 that disk IO was the dominant cost factor of query evaluations. In our experiments, 60 to 90 percent of total elapsed time was spent on disk access by the \mathcal{EE} -Join algorithm.

Performance of Processing Paths with Fixed Length As mentioned in Section 4.3.2, using the depth information stored with each element, the \mathcal{EE} -Join algorithm can process element-element join queries of fixed path lengths, e.g., parent-child type basic queries. In this set of experiments, we examined the performance of \mathcal{EE} -Join queries with different path lengths. These queries were in the following

Data Set	E_A	E_B			
		E_A/E_B	$E_A/*/E_B$	$E_A/**/E_B$	$E_A//E_B$
Shakespeare	ACT	SCENE	SPEECH	LINE	LINE
SIGMOD	articles	articlesTuple	authors	author	author
NITF-100	body.content	block	block	block	block
NITF-1	body.content	block	block	block	block
DBLP	dblp	inproceedings	author	N/A	author

TABLE 4.4. Summary of \mathcal{EE} -Join Queries of Different Path Lengths

form: E_A/E_B (length one), $E_A/*/E_B$ (length two), $E_A/**/E_B$ (length three) and $E_A//E_B$ (length unknown). Note that the asterisk, *, in a path matches any (single) element. The actual queries of different path lengths for different data sets are shown in Table 4.4.

Figure 4.5 shows the performance results from the \mathcal{EE} -Join queries, measured in the speed-up obtained by the \mathcal{EE} -Join algorithm over the bottom-up method. That is, the y-axis represents the ratio of the elapsed time consumed by the the bottom-up method to the elapsed time by the \mathcal{EE} -Join algorithm. For the DBLP data set, we used the conference publications portion of the DBLP bibliography. Since the longest path length was two in the DBLP data set, there is no performance result from the queries of path length three (*i.e.*, $E_A/**/E_B$) for the DBLP data set in Figure 4.5. From the results, it is evident that the \mathcal{EE} -Join algorithm outperformed the bottom-up method significantly for all different path lengths and for all five data sets.

Performance of \mathcal{EA} -Join The queries we used for element-attribute join operations are of the form $E[@A]$. For example, the path expression, `figure[@caption]`, is to find all `figure` elements with a `caption` attribute. For this type of queries, we compared the performance of the \mathcal{EA} -Join algorithm with both top-down and bottom-up methods. The actual queries used in the experiments are summarized in Table 4.5. The Shakespeare data set was not used, because the data set contains no

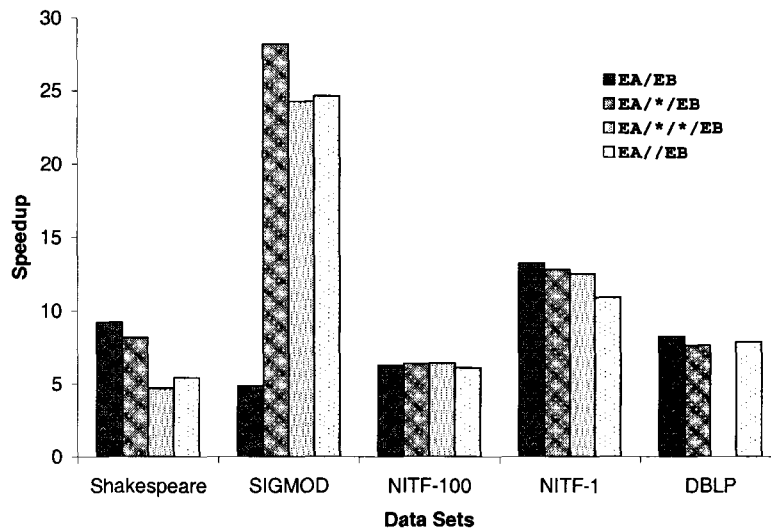


FIGURE 4.5. Speed-up of $\mathcal{E}\mathcal{E}$ -Join over Bottom-up for Different Path Lengths

Data Set	Element E	Attribute A	# of E 's	# of A 's	# of $E[@ A]$'s
SIGMOD	author	id	9836	8934	6099
NITF-100	block	dir	3476	25757	2649
NITF-1	block	dir	2801	17152	2127
DBLP	inproceedings	key	140936	142116	140936

TABLE 4.5. Summary of $\mathcal{E}\mathcal{A}$ -Join Queries

attributes.

The total elapsed and IO time of the SIGMOD and two synthetic data sets are shown in Figure 4.6 and Figure 4.7 respectively. For a data set with a relatively small number of attributes such as the SIGMOD data set, the bottom-up method was expected to outperform the top-down method, because traversing up the tree for a small number attributes can be more efficient than traversing down the tree with many branches. In our experiments, for the SIGMOD data set, the bottom-up method was the best, followed by the $\mathcal{E}\mathcal{E}$ -Join algorithm very closely, then by the top-down method in Figure 4.6(a) and Figure 4.7(a).

For synthetic data sets, however, the number of attributes was much larger than the number of elements. For such data sets, the performance of the bottom-up method

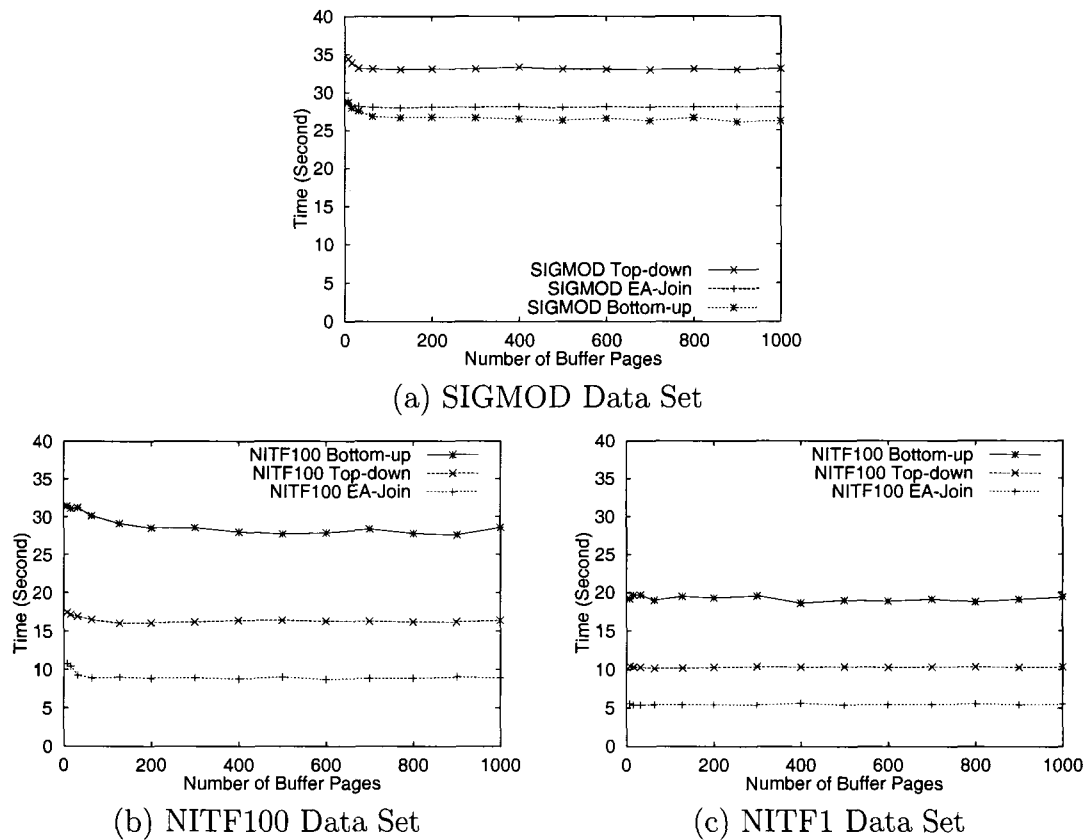


FIGURE 4.6. Total Elapsed Time of Query E[@A]

degenerated substantially, because it had to look up the parent elements for so many attributes. We can see this from Figure 4.6(b)-(c) and Figure 4.7(b)-(c). Since attributes are not allowed to have child nodes, the scope of traversal from an element to its child attributes is limited to one level of a tree. Thus, the top-down method was fairly efficient for the synthetic data sets. Nonetheless, the performance of the $\mathcal{EA}\text{-Join}$ algorithm was still better than the top-down method. The main reason is that $\mathcal{EA}\text{-Join}$ needs to scan the element list and attribute list only once without traversing trees.

Figure 4.8 depicts the speed-up achieved by the $\mathcal{EA}\text{-Join}$ algorithm over the top-down and the bottom-up methods. As we pointed out earlier, the bottom-up

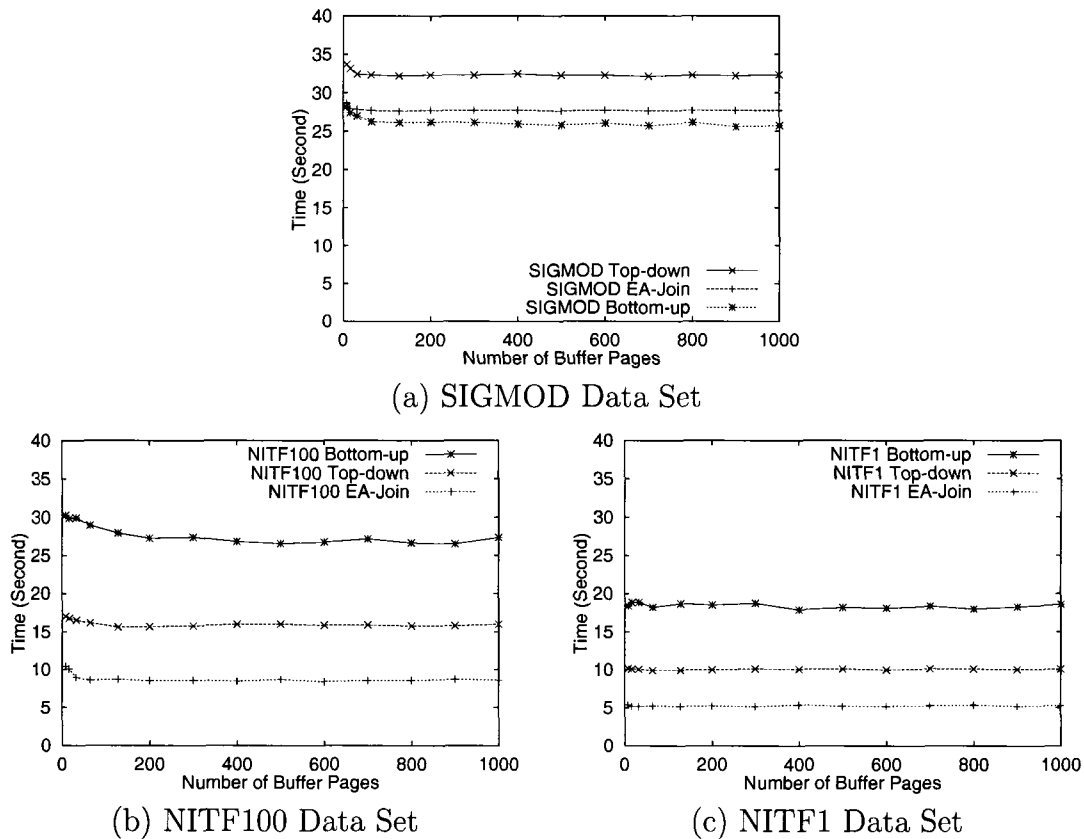


FIGURE 4.7. IO Time of Query E[@A]

method was the best for the SIGMOD data set and slightly faster than the $\mathcal{EA}\text{-Join}$ algorithm. The results from the two NITF data sets showed a similar trend, because the data sets were generated based on the same DTD. For the DBLP data set, we used `inproceedings` element and `key` attribute for the element-attribute join experiment. Since the `key` was the only attribute of `inproceedings` in the conference portion of the DBLP data set, the number of `inproceedings` elements was almost identical to the number of `key` attributes. For the reason, the bottom-up and the top-down methods yielded almost identical performance. Nonetheless, both of them spent more than twice the time spent by the $\mathcal{EA}\text{-Join}$ algorithm.

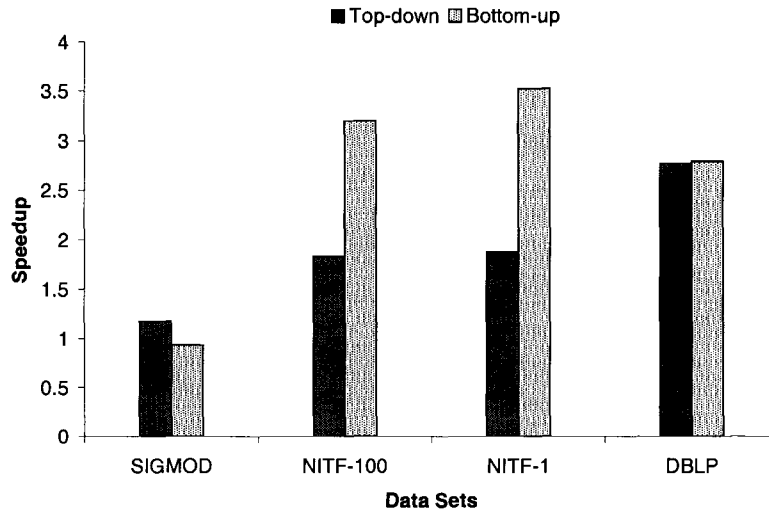


FIGURE 4.8. Speed-up of \mathcal{EA} -Join over Top-down and Bottom-up

Scalability Test We carried out scalability tests of the proposed algorithms with a large synthetic data set. This data set was generated using the XML Generator and the NITF document type definition. There were 100 files in this data set with a total size of 229 MB. This experiment was performed with a varying number of files from 5 to 100 with an increment of 5. We could also use a single XML document for each data size test. For example, we could combine a set of small files into a large one by using a common root element. However, the performance results would be almost the same, since the order of all $(did, order)$ pairs remain the same. The queries used in this experiment were the same as those used for \mathcal{EE} -Join and \mathcal{EA} -Join experiments with the NITF data sets. In Figures 4.9(a) and 4.9(b), for both element-element and element-attribute joins, the query processing time increased almost linearly, as the size of XML data increased. This result shows the linear scalability of the proposed algorithms, and provides another evidence that the proposed sort-merge based algorithms can improve the performance of query processing for XML path expressions over conventional methods by up to an order of magnitude.

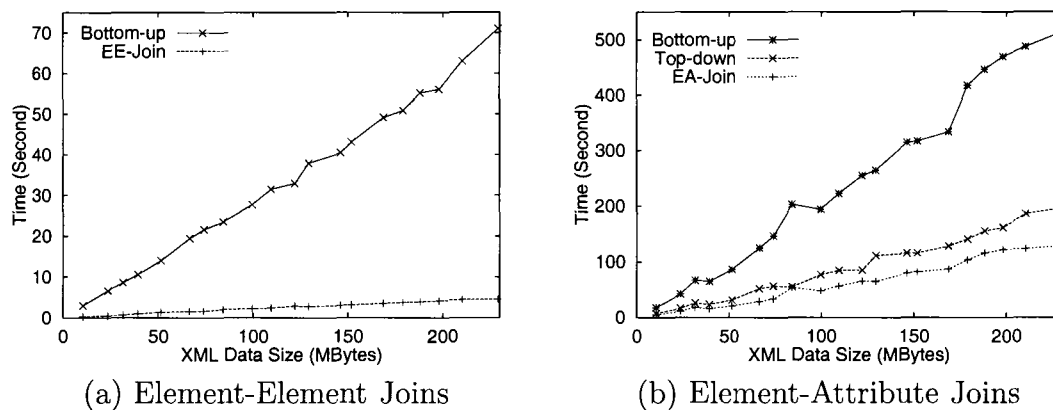


FIGURE 4.9. Scalability Test of $\mathcal{E}\mathcal{E}$ -Join and $\mathcal{E}\mathcal{A}$ -Join

4.4 Partition-Based Algorithms

Sort-merge based algorithms assume the inputs are in sorted order of assigned numbers, but this order is not always guaranteed. For example, an input may be sorted by data values, or it may be the result from operations using hash indexes. In order to use sort-merge based algorithms, we need to sort input data beforehand, which could be costly. Graefe *et al.* compared the concepts behind sort- and hash-based query processing algorithms [33] and concluded that:

- many dualities exist between the two types of algorithms,
- their costs differ mostly by percentages rather than factors,
- several special cases exist that favor one or the other choice, and
- there is a strong reason why both hash- and sort-based algorithms should be available in a query processing system.

Partition-based algorithms can be considered as hash-based algorithms. We expect that they should also be available for path query processing. In this section, we discuss the use of partition-based algorithms to process XML ancestor-descendant

join queries. Guided by Graefe’s findings, we will identify the cases that will favor partition-based or sort-merge based algorithms.

We proposed several partition-based algorithms. When the descendant input is used as the outer set in the join, the *Descendant Partition Join* algorithm can be used to process join operations. When the ancestor input is used as the outer set, we propose the *Segment-tree Partition Join* and the *Ancestor Link Partition Join* algorithms. The experimental results show that the Ancestor Link algorithm can make the best use of memory buffer and take advantage of unevenly sized inputs. We believe that these algorithms are important choices for query optimizers to consider during XML query processing for different input characteristics.

In the rest of this section, we will first introduce the data partitioning and related techniques. Then three partition-based algorithms are proposed. After that, we show the performance study of the proposed algorithms.

4.4.1 Data Partitioning

The first step in partition join is to partition both input data sets that need to be joined. Each pair of corresponding partitions from both sets need to be joined. In order to avoid data re-reads, at least one partition of the pair should fit in memory. If no information about the data distribution is available, sampling data can help to determine the partition boundaries. There has been some research work addressing the random sampling problem [14, 48]. We have chosen a similar algorithm to “determinePartIntervals” algorithm [60] to determine the partition intervals. This partition algorithm considers partition cost, sampling cost and join cost to minimize the total I/O. The cost of sampling is computed based on the Kolmogorov test statistic [22]. During sampling, we clustered disk page reads such that sample pages close enough are read sequentially instead of several random reads. If one partition is larger than memory size, further partitioning can be applied recursively. In this dissertation,

we assume that each outer partition can fit in memory, which is also true in our experiments.

Unlike the point set, for a range set, there is a possibility that some ranges can overlap with multiple partitions, no matter how the partition boundaries are determined. Which partition should we put these long ranges in? A straightforward solution is to replicate the ranges to all partitions it overlaps. This requires additional disk I/O to handle replicates. Instead, we adopted the solution proposed in [60], where the partitioning is performed using only the start point of each range. A range is put in the partition, where the start point of the range falls in. One requirement of this method is that the partition join should be evaluated in increasing order of partitions. When the current partition is done, the ranges that cross the next partition boundary are kept in memory cache. These ranges in cache will participate in the join of the next partition.

One nice property of the XML data range set is that the number of ranges crossing any partition boundary is bounded by the height of the document tree. Thus, we can pre-allocate the in-memory cache to hold those crossing boundary ranges according to the tree height. A possible improvement is to use the maximum level difference of the ranges plus one as the cache upper bound. This upper bound can be obtained before sampling and partition. We can take the size of this in-memory range cache out from the total memory buffer size before partitioning. So, the partitioning can be done as normal point data partitioning without considering the boundary crossing problem.

4.4.2 Descendant Partition Join

In the *descendant partition join* algorithm, which is shown in Algorithm 3, the descendant point set is the outer set. The sampling and partitioning is based on the point set. During the join phase, each point partition is loaded in memory to be

Algorithm 3: Descendant Partition Join

Input: (ancestor range set, descendant point set)

- 1 Set range cache to be empty;
- 2 Determine partition boundaries according to the descendant point set;
- 3 Partition both range and point sets;
- 4 **for** *Each partition pair in increasing order* **do**
- 5 Load descendant partition in memory;
- 6 **for** *Each range in range cache* **do**
- 7 Join the range with descendant partition;
- 8 Remove the range if it doesn't cross the next partition boundary;
- 9 **end**
- 10 **for** *Each page of range partition* **do**
- 11 Load the range set page in memory;
- 12 **for** *Each range in the page* **do**
- 13 Join the range with descendant partition;
- Put the range in the range cache if it crosses the next partition boundary;
- end**
- end**
- end**

joined with the corresponding range partition and the cached ranges from previous range partitions.

In line 7 and line 12, the join operation is to join one range with all the descendant points in a partition. Since it is an in-memory operation, at first, we directly used the nested loop join. From our preliminary experimental results, we found that the performance of the nested loop join was very bad due to the high CPU and memory access cost. So, we changed the nested loop to binary search. The descendant point partition is sorted after it is loaded in memory. When a range is to be joined with the points partition, a binary search is used to locate the first point in the range. Then, we scan the sorted point set until we reach the last point in this range.

In line 13, after the join of each range, we put the range in the range cache if it crosses the next partition boundary. At the beginning of the join, the ranges in the range cache are joined with points first (see line 7). At the same time, we try to eliminate the ranges that do not cross the next partition boundary, which is shown

Algorithm 4: Segment Tree Partition Join

Input: (ancestor range set, descendant point set)

- 1 Set range cache to be empty;
- 2 Determine partition boundaries according to the ancestor range set;
- 3 Partition both range and point sets;
- 4 **for** *Each partition pair in increasing order* **do**
- 5 Load ancestor range partition in memory;
- 6 Build a segment tree for ranges in the partition and the range cache;
- 7 **for** *Each page of point partition* **do**
- 8 Load the point set page in memory;
- 9 **for** *Each point in the page* **do**
- 10 Join the point using the segment tree;
- end**
- end**
- 11 Dispose the segment tree;
- 12 Remove from range cache the ranges not crossing the next boundary;
- 13 Put ranges crossing the next boundary from range partition to range cache;
- end**

in line 8. Thus, the range cache is maintained in such a way that only the ranges crossing the next partition boundary are kept in memory, and are used in the join of the next partition.

In a partition join, either join input data set can be chosen as the outer set to determine the partition boundaries. However, partitioning based on the larger input produces more partitions than partitioning based on the smaller input. With the smaller input set as the outer set, we can have smaller number of partitions and more sequential I/O. If the smaller input can totally fit in memory, there is no partitioning needed at all. Usually, in XML data, the size of the descendant point set is larger than that of the ancestor range set. So, partitioning based on the ancestor range set could produce better performance. We next introduce two partition join algorithms using the ancestor range set as the outer set. They are named *Segment Tree Partition Join* and *Ancestor Link Partition Join* based on their in-memory join algorithms.

4.4.3 Segment Tree Partition Join

In partition join algorithms using the ancestor range set as the outer set, the partitioning is based on the ancestor range set. As in the *descendant partition join* algorithm, a range cache is also used. During the join phase, each range partition is first loaded into memory. These ranges in the partition and the ranges in the range cache are together to be joined with each descendant point from the inner partition. Because the nested loop join is inefficient, in the segment tree partition join algorithm, which is shown in Algorithm 4, the segment tree [55] is used as the in-memory algorithm to quickly find a set of ranges containing a point. The worst case space complexity of the segment tree is $O(N \cdot \log N)$, where N is the number of ranges [55]. In our implementation, we used several techniques to minimize the size of the segment tree data structure. In this in-memory join context, the ranges and their end-points are already known. No insertion and deletion of end-points is required after the tree is built. We can utilize this property to build a static segment tree, which is more compact than a dynamic one.

The first step of building a segment tree is to sort the end points of the ranges in memory (duplicate points should be eliminated). Let us suppose the number of ranges is N . Because the start point of each range is a unique preorder ID number, there are at least N points in the segment tree. After sorting, an *virtual* empty segment tree is there, because the parent-child relationship in the segment tree can be determined by the array index calculation. There is no additional space needed for the structure pointers of the tree. However, the number of segment tree nodes is twice the size of the point array. When a range is inserted to the segment tree, a pointer is needed to store the linked list for each segment tree node. So, at least $2N$ pointers are needed. For each range, at least one linked list record is associated with it. Each linked list record contains the range ID and the next link pointer. In total, we need at least $3N$ pointers, and N range ID's. If pointers and ID's are represented by integers,

then at least $4N$ integers are required for the segment tree. This is a large memory requirement, since in our implementation, each XML element node record only uses four integers. In this case, half of the memory is used for the segment tree.

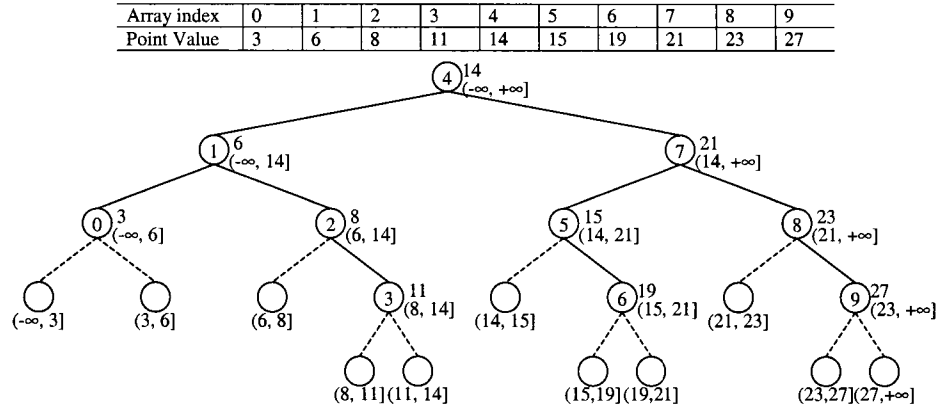


FIGURE 4.10. End Point Array and Its Segment Tree

As an example, the table in Figure 4.10 shows a set of point values along with their index in an array. In the segment tree illustrated below, the number in a node circle is the index value. It is empty for leaf nodes, since they are virtual nodes. The point value and the node range are labeled near each node. The root index value of each sub-tree is the middle index value of the index range of the sub-tree. That is $tree\ root = \lfloor (start\ index + end\ index) / 2 \rfloor$. For example, index value 4 is the root of the index range $[0, 9]$, which is the whole tree.

Since memory size is limited, the more memory used for in-memory index (such as the segment tree), the smaller memory is left for partitions. To solve the large memory occupation problem of the segment tree, we propose to use *Ancestor Link* algorithm for in-memory join, which is described in the next section.

4.4.4 Ancestor Link Partition Join

The control flow of the *Ancestor Link Partition Join* algorithm is the same as the segment tree partition join algorithm (Algorithm 4), except that the in-memory join

is now the *ancestor link join*. Instead of building a segment tree, an ancestor link data structure is used. Each descendant point is joined with all the in-memory ranges using this ancestor link data structure.

According to the *range containment property*, for any two ranges, either one range contains the other range, or they are disjoint. We can build a *range tree* according to the containment relationship. In the range tree, a child range is directly contained in the parent range, which is the smallest range containing the child. If a descendant point p is contained in a range R in the range tree, then this point p is also contained in all the ancestor ranges of range R in the range tree. Using this range tree, we can efficiently perform the join between a point and a set of ranges.

Since all range records to be joined are already in the buffer memory, no additional range information needs to be duplicated again in the range tree. In our implementation, the range tree is only a pointer array, which has the same size as the number of the ranges in memory. The positions of pointers correspond to the positions of the ranges in the range array. The content of a pointer is the array index of the parent range. So, an edge in the range tree is pointing from a child node to a parent node. To build the range tree, we first sort all the ranges by the start points. After sorting, the ranges are also in sorted preorder with respect to the range tree. Then we scan the range array once with a pointer stack to find the parent of each range and update the ancestor pointer array. This algorithm is shown in Algorithm 5.

Next, we show how to find the first (smallest) range containing a given point p in the range tree. Since the range array is sorted by the start point, we can use binary search to find the range R_p that may contain the point p . That is R_p is the first range whose start point is smaller than p . If R_p contains p , then all the ancestors of R_p in the range tree also contain p . If R_p does not contain p , we can follow the ancestor link of R_p to find the ranges that contain p . The main advantage of the ancestor link is its small memory requirement. The whole structure is only an array of pointers. The size is at least less than one fourth of the segment tree. So, more memory can

Algorithm 5: Build Ancestor Link

Input: (sorted range array, range tree array)

- 1 Initialized pointer stack to be empty;
- 2 **for** *Each range in the range array* **do**
- 3 **while** *pointer stack is not empty and the top of pointer stack is not the parent of the current range* **do**
- 4 Pop the pointer stack;
- 5 **end**
- 6 **if** *pointer stack is empty* **then**
- 7 Set the current range tree pointer to be null;
- 8 **else**
- 9 Set the current range tree pointer to be the top of the pointer stack;
- 10 **end**
- 11 Push the current range pointer to the pointer stack;
- 12 **end**

be used for partitions.

4.4.5 Performance Study of Partition-Based Algorithms

We implemented these three partition join algorithms discussed in the previous sections. They are referred to as *partition-d*, *partition-s* and *partition-a* for Descendant partition join, Segment-tree partition join and Ancestor-link partition join algorithms respectively. For the performance study, we also implemented the sort-merge based algorithms similar to the Stack-Tree join algorithm (Stack-Tree-Desc) [61]. Two variations of sort-merge join algorithms, *sortmerge-s* and *sortmerge-m*, were implemented. *Sortmerge-s* sorts each input into a single run before the join phase, while *sortmerge-m* combines the last merge phase of sorting with the join phase.

The above algorithms were all implemented on top of a paged file layer, which also provides buffer management functionality. In our experiments, the page size was 4K bytes. The input files were paged files, and were directly generated from the data-set in random order. We have chosen two data sets, Shakespeare and DBLP, to demonstrate the performance. For the DBLP data set, we used the conference

Data Set	E_A	E_B	#of E_A	#of E_B	E_A Page#	E_B Page#
Shakespeare	ACT	SPEECH	185	31028	1	122
	SPEECH	LINE	31028	107833	122	423
DBLP	dblp	author	783	320300	4	1257
	inproceedings	author	140936	320300	553	1257

TABLE 4.6. Queries for Performance Study

portion with a raw data size 58MB.

In the experiments, we measured the elapsed time (both CPU and IO) of query processing. For fair comparison, the sorting cost for sort-merge based algorithms, and the sampling cost for partition based algorithms were both counted in the performance measure. Since the cost of output generation is the same regardless of algorithms applied, the output cost was not included in the measurements. Experiments were performed on an Intel workstation with a Pentium 4 1.6GHz CPU running Solaris 8 for Intel platform. This workstation has 512M bytes of memory and a 40GB EIDE disk drive (with 7200 RPM and 8ms average seek time). The disk is locally attached to the workstation and used to store XML data. We used the direct I/O feature of Solaris for all experiments to avoid operating system's cache effects.

Table 4.6 describes the queries we used in the experiments. It also provides the ancestor range set size and the descendant point set size information. All queries are of the form " $E_A//E_B$ ", which is to find the ancestor-descendant element pairs. We have chosen these queries on different element list sizes that can enable us to show the performance differences between sort-merge based and partition-based algorithms.

Figure 4.11 shows the performance for two queries on the Shakespeare data and Figure 4.12 shows the performance for the DBLP data. The performance measure is elapsed time in seconds with different number of memory buffer pages. Although the size of the DBLP data is much larger than that of the Shakespeare data, we observed similar performance trends.

In Figure 4.11(a) and Figure 4.12(a), the ancestor set size is small. Since the

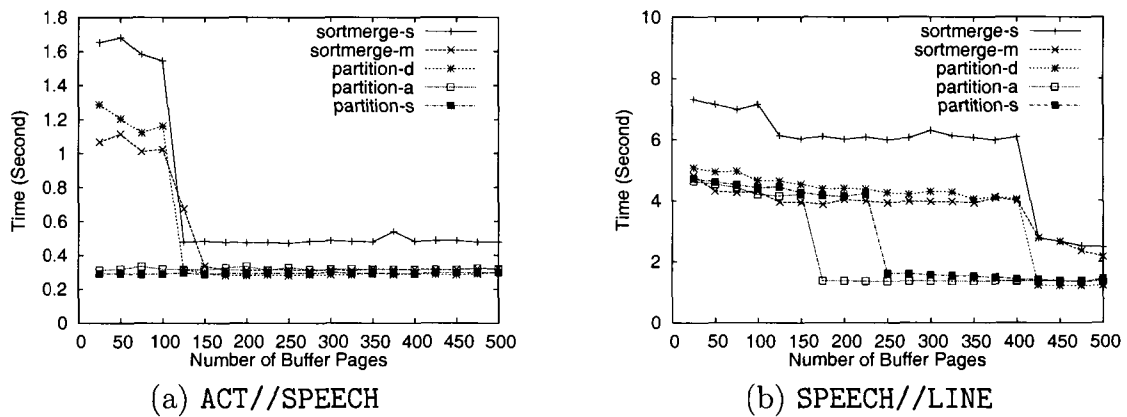


FIGURE 4.11. Shakespeare Data Queries

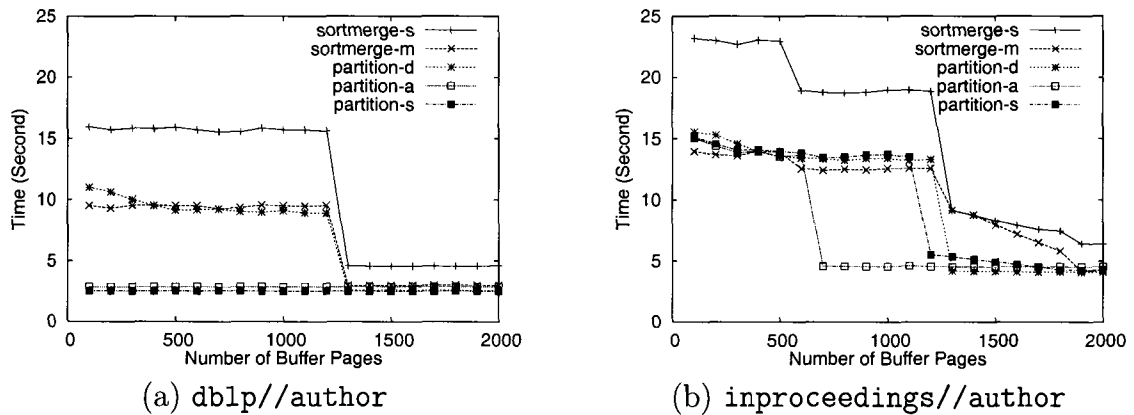


FIGURE 4.12. DBLP Data Queries

partition-a and the partition-s algorithms use the ancestor set as the outer set, there is no need to do partitioning at all if the memory can hold the outer set. So, their performances are improved when the total buffer size is small.

For both sort-merge join and partition join algorithms, if the memory buffer can hold both sets, there is only one scan of the input files needed to load data in memory. That is the minimum cost for the query processing. So, when the memory buffer size is large enough, there is a sharp performance increase. After this, the performance remains almost constant. This trend can be seen from Figure 4.11(a)

and Figure 4.12(a). The performance increases correspond to the input data sizes. For example, in Figure 4.12(b) when the partition-a algorithm time drops, the buffer pages can hold the ancestor set, which means the ancestor is about the same as the buffer pages. This also conforms to the page number information in Table 4.6. From the table, we know that the `inproceedings` element has 553 pages, which is about the location where the time of the partition-a algorithm drops in Figure 4.12(b).

In the `sortmerge-m` algorithm, the last merge phase of sorting is combined with the join phase, so the last merge scan is saved. On the other hand, for the `sortmerge-s` algorithm, the last sorting merge scan of input files is always needed. It increases both the I/O and the CPU cost. Among these algorithms, the `sortmerge-s` algorithm is the worst, which is clearly shown in Figure 4.11 and Figure 4.12.

In Figure 4.11(b) and Figure 4.12(b), the ancestor set is large. When neither join set can fit in memory, all algorithms have to scan the input files at least twice. In this case, the partition join algorithms (`partition-d`, `partition-a` and `partition-s`) and the `sortmerge-m` algorithm have similar performance. From this result, we can also see that the sampling cost for the partition-based algorithms is low. The reason is that the number of samples is small compared to the whole set, and we also optimized the sampling process using clustered read technique as described in Section 4.4.1.

With the memory size getting larger, the performances of the partition join algorithms improve faster than the `sortmerge-m` algorithm, since the partition join algorithms can avoid partitioning when one join set can fit in memory. For the `sortmerge-m` algorithm, it can keep all runs in memory only if the memory is large enough to hold both join sets. Otherwise, additional I/O for runs is unavoidable. Among the partition join algorithms, the performance of the `partition-a` algorithm is the best. The in-memory join data structure *ancestor link* requires less memory than *segment tree*. So, more memory can be used to hold partitions.

The above experimental results provide useful information for a query optimizer to choose suitable algorithms for different inputs. If both inputs are not in sorted

order and the smaller input can be fully fit in memory while memory cannot hold both inputs, partition-based algorithms are clearly a better choice. A query optimizer can choose the smaller set as the outer set. If the smaller set is the ancestor set, the partition-a algorithm is better. The reason is that the segment-tree structure is always larger than the Ancestor Link array. Thus, the performance of partition-a is always better than partition-s. If the smaller set is the descendant set, we can choose the partition-d algorithm. On the other hand, for the following cases sort-merge based algorithms are preferred.

1. Both inputs are in sorted order.
2. Join results are large and need to be in sorted order for future processing.

If both inputs can be fit in memory or none of them can be fit in memory, the differences between sort-merge and partition-based algorithms are not dramatic. We can utilize a cost model to estimate different costs and make the best decision. We believe that the performance study in this section provides useful information to build the cost model, which is an important future work.

4.5 Summary

For XML data, with the introduction of *the extended preorder numbering scheme*, XML path queries can be processed using traditional relational database techniques, e.g., sort-merge based algorithms and partition-based algorithms.

In this chapter, we first introduced the XML Indexing and Storage System (XISS) to store and index XML data and to efficiently process regular path expression queries. The major drawback of the conventional methods based on tree traversals is that they may often require an extensive search of XML data trees. To avoid this drawback, we have proposed an innovative approach to processing a path expression query, which decomposes a complex path expression into a collection of basic path subexpressions.

Each subexpression can be processed either by directly accessing index structures of the XISS system or by applying one of the proposed \mathcal{EA} -Join and \mathcal{EE} -Join algorithms. For a subexpression having a pair of elements, for example, \mathcal{EE} -Join algorithm performs its processing by a two-stage sort-merge operation. Experimental results from our prototype implementation of XISS show that the proposed algorithms can achieve performance improvement over conventional methods by up to an order of magnitude.

We also proposed the partition-based algorithms, which can be chosen by a query optimizer according to the characteristics of the inputs. An in-memory range cache was used to hold ranges crossing partition boundaries. In the Ancestor Link partition join algorithm, we proposed to use the *Ancestor Link* data structure, which is much smaller in size compared to the segment-tree. So, more memory could be used for holding partitions. The experimental results showed that the Ancestor Link algorithm could make the best use of memory buffer and take advantage of the uneven sized inputs. We believe that these algorithms are important choices for a query optimizer to consider during XML query processing.

CHAPTER 5

XML TWIG QUERY PROCESSING

5.1 Introduction

As we have discussed before, XPath provides path expressions to navigate through the document structure and locate XML data. As a sub-language of XQuery, XPath expressions are commonly used in XQuery. For example, the XQuery example of Figure 1.3 in Chapter 1 has three path expressions, which are `/books/book`, `$b/price`, and `$b/title`. Further more, the XPath specification defines quite a few addressing mechanisms, and they can be combined in many different ways. As a result, XPath delivers a lot of expressive power for a relatively simple specification. For example, we can use the following path expression to express the same query in Figure 1.3:

```
Q2: /books/book[price > 100][title = "Expensive Book"]
```

In Q2, there are two predicates, enclosed in two pairs of square brackets, to filter the `book` elements. As the result of Q2, a sequence of `book` elements are returned. Because of the introduction of predicates, Q2 is not just a single path. Each predicate in Q2 forms a branch, which makes the query to be a tree pattern. Figure 5.1 shows this tree pattern of Q2. Each node in this figure is an element query node, which matches the list of elements with the same name as the node during query processing. Note that in the figure the `book` node is the *target node*, which is to be output and is highlighted with a black dot. Since a query such as Q2 is a tree patterned path query and it matches all twigs in document trees, we refer this kind of path queries as *twig queries* in this dissertation.

With the introduction of the extended preorder numbering scheme on XML data, a path expression can be processed by basic structural join algorithms introduced in

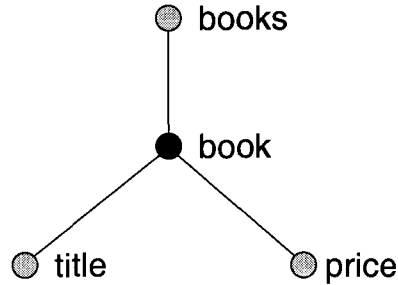


FIGURE 5.1. The Tree Pattern of Q2

Chapter 4. To process a complex query such as Q2, we can decompose it into a set of basic structural relationships: ancestor-descendant and parent-child relationships. For example, Q2 can be decomposed to four basic structural joins, **books/book**, **book/price**, and **book/title**. After the decomposition, the basic parent-child and ancestor-descendant type structural joins can be processed and merged together. For example, we can first get all **book** elements under the **books** element. Then this intermediate **book** element list is filtered by the **price** and **title** elements. In the end we get the qualified **book** elements. A problem of decomposing a query and using pairwise joins is that the intermediate results could be large. For example, the join result from **books** and **book** could be very large. But the final result of the whole query may be very small. It is costly to generate and access large intermediate results during query processing. In this chapter, we propose the *IndexTwig* algorithm to process twig queries as a whole and avoiding generating intermediate results.

There are two twig query processing algorithms proposed in the literature. Bruno *et al.* [10] proposed a holistic twig join algorithm, called TwigStack, for matching an XML query twig pattern without decomposing the query into basic structure joins. The algorithm can avoid generating the intermediate results of basic structure joins, which can be potentially large. They also showed the TwigStackXB algorithm, which uses XB-trees (a modification of B-trees) to index elements. The TSGeneric+ algorithm [41] is a refinement of the TwigStack algorithm. The TSGeneric+ algorithm

improves the TwigStack algorithm by introducing the *SJCursor*, which uses the XR-tree index [40] to skip both ancestor and descendant data.

Previously proposed twig query processing algorithms output all matched patterns of a twig query. Based on the semantics of the XPath expressions, in this chapter, we propose a simplified output model. In this model, only the answers of the *target* query node are output. We also propose the *Containment B⁺-tree* (CB-tree) index, which is an extension of the traditional B⁺-tree, to support both the *containment query* and the *reverse containment query* (refer to the next section for definitions). Based on the new output model and the index, the *IndexTwig* algorithm is introduced. It can process a twig as a whole without generating intermediate results. It also skips unnecessary data by using the CB-tree.

We reiterate the contributions of this chapter in the following.

- The proposed CB-tree index can support both *containment query* and the *reverse containment query*. The CB-tree is effective for XML documents with or without a small number of recursions, When there are no recursively nested elements (in which an element can be a sub-element of itself) in XML data, the CB-tree is the same as the B⁺-tree. Commercial database vendors can implement this technique without much modification to the traditional B⁺-tree index.
- Based on the simplified output model, the IndexTwig algorithm enables output optimizations that can skip unnecessary data and output. It is also a flexible algorithm that can work with any index supporting containment and reverse containment queries. It works with the conventional B⁺-tree when there is no recursive nesting.
- We have developed the *Fast Existence Test* (FET) optimization technique for the IndexTwig algorithm, which skips unnecessary data access and output, and avoids generating unnecessary results.

- The experimental results demonstrate that the IndexTwig algorithm with the FET optimization is highly efficient to process twig queries. It outperforms the TSGeneric+ algorithm by a large factor in many cases.

The rest of the chapter is organized as follows. We introduce related work in Section 5.2. In Section 5.3, we describe the CB-tree index structure. The proposed IndexTwig join algorithm and FET optimization are introduced in Section 5.4. In Section 5.5 we present the performance study. In the end, we conclude the work of this chapter in Section 5.6.

5.2 Related Work

Much work has been done in the area of indexing and querying XML data. In this section, we present the related work in two subsections. The first subsection describes indexing techniques. The second subsection focuses on path expression processing algorithms.

5.2.1 XML Indexing Techniques

We divide the indexing techniques into two categories, structural indexes and sequence-based indexes. A structural index provides a structural summary or index of the existing data. Among the structural indexes, a navigational index can be used in traversal based query processing. Since the tree structured data model can be considered as a special case of graph-based data models, the techniques used in graph-based data models can be applied on XML data. Along with the introduction of numbering schemes to encode XML data, several new structural indexing structures have been proposed. An XML numbering scheme keeps the ancestor-descendant information by assigning numbers to each node in the XML hierarchy. Indexes are built to help access the encoded elements and determine relationships during query processing.

In sequence-based indexes, the XML data are first encoded into a long sequence. Using the same encoding method, the query is converted into a subsequence. The query processing is reduced to a subsequence search problem. The sequence indexes are built on the sequence data and help the subsequence search.

Structural Indexing For XML databases with graph-based data models, path traversals play a central role in query processing, and using indexes to help navigation is an important issue. The Lore [56, 50] query optimizer stores statistics about all possible sub-paths (*i.e.*, label sequences) in the database up to a certain length k . To speed up query processing in a Lore database, four different types of index structures have been proposed [32, 52].

Cooper *et al.* proposed an index structure called *index fabric* for XML path expression queries [18]. All the root-to-leaf element paths in XML data trees are inserted into a disk-resident Patricia trie as strings. The key idea of the index fabric is to transform a memory-based Patricia trie into a block-structure counterpart, so that it can store and access the disk-resident element paths efficiently. Covering indexes [43] were proposed to build small index structures for branching path queries. Suciu *et al.* proposed the T-index structure that maintains equivalence classes of document nodes which are indistinguishable with respect to a given path template.

For numbering scheme based indexes, Chien *et al.* [15] used the B⁺-tree and its enhancement to expedite a structural join. For a given tag, a corresponding *Containment Forest* is embedded in a B⁺-tree with element's parent and sibling pointers stored in leaf records. By using these additional pointers, the sort-merge join algorithm can avoid unnecessary index traversals. The sort-merge join based on the *synchronized tree traversal* using the R-tree is also discussed in the paper.

Bruno *et al.* introduced a variant of the B⁺-tree, the XB-Tree [10]. The entries in the leaf pages of the XB-tree are sorted by the element ID, which is similar to a B⁺-tree. Each entry in the internal pages maintains a *bounding segment* that com-

pletely covers all the element ranges indexed under this entry. In this way, some index pages can be skipped by checking this bounding segment. Each page also has a parent pointer pointing to its parent page, which enables the index access to go up toward the root.

Jiang *et al.* [40] proposed the XR-tree that can support the reverse containment search. The XR-tree stores copies of element ranges in *stab lists* pointed by internal nodes in the internal nodes. Through our investigation, we noticed that in real word applications, documents with very deep recursions are scarce. We also observed that in B⁺-tree indexes, the space utilization is not 100 percent, which means we can make use of the empty space inside index pages to store replicated elements. This motivated us to propose the CB-tree index, which will be introduced in Section 5.3.

Sequence-Based Indexing By encoding XML data and path queries as sequences, the query processing is reduced to a subsequence search problem.

Wang *et al.* have proposed a method called ViST that transforms XML data trees and twig queries into structure-encoded sequences [63]. By representing both XML documents and XML queries in structure-encoded sequences, querying XML data is equivalent to finding subsequence matches. The structure-encoded sequence is a two-dimensional sequence of (symbol, prefix) pairs $\{(a_1, p_1), (a_2, p_2), \dots, (a_n, p_n)\}$ where a_i represents a node in the XML document tree, and p_i represents the path from the root node to node a_i . The nodes a_1, a_2, \dots, a_n are in preorder. ViST performs subsequence matching on the structure-encoded sequences to find twig patterns in XML documents. These sequences are stored in a trie.

Rao *et al.* proposed to transform an XML document tree into a sequence by Prüfer’s method that constructs a one-to-one correspondence between trees and sequences [57]. During query processing, a twig pattern is also transformed into its Prüfer sequence. Then all the occurrences of the twig pattern in the database can be found by performing subsequence matching on the set of sequences in the database,

and performing a series of refinement phases,

5.2.2 XML Path Processing Algorithms

For XML databases using graph-based data models, path traversals play a central role in query processing, and optimizing navigational path expressions is an important issue. The optimal query plan depends not only on the *values* in the database but also on the *shape* of the graph containing the data. Three query evaluation strategies have been proposed for Lore’s cost-based query optimizer [51]. They are a top-down strategy for exploiting the path expression, a bottom-up strategy for exploiting value predicates, and a hybrid strategy.

The problem of optimizing regular path expressions has been studied in the context of navigating semi-structured data in web sites [4, 27]. The semi-structured data is modeled as an edge-labeled graph, where nodes denote HTML pages and edges denote hyperlinks. Abiteboul and Vianu [4] deal with a path query evaluation that takes advantage of local knowledge (*i.e.*, path constraints) about data graphs that may capture structural information about a web site. They address the issue of equivalence decidability of regular path queries under such constraints. Fernández and Dan Suciu [27] proposed two query optimization techniques to rewrite a given regular path expression into another query that reduces the scope of navigation.

With the use of numbering schemes to encode XML data, various indexes and algorithms [10, 15, 47, 68, 61] have been proposed for the processing of structural joins. The stack based algorithms [61] utilize stacks to keep elements in-memory to avoid data re-reads. Bruno *et al.* [10] proposed the TwigStack algorithm, which can process an XML query twig pattern without decomposing the query into basic structural joins. Also in their work, the XB-Tree [10] was introduced as a variant of the B⁺-tree to work with the TwigStack algorithm.

Chien *et al.* [15] used the B⁺-tree and its enhancement to expedite a structural

join by avoiding elements that do not participate in the join. The focus there was on basic structural joins. Since the index did not support the reverse containment query, the ancestor skipping was not fully exploited. The index and algorithm worked well for documents with highly nested structures. For relatively flat structure, the opportunities to skip ancestors using sibling pointers were limited.

The *TSGeneric⁺* algorithm [41] was proposed based on the TwigStack algorithm. The *TSGeneric⁺* algorithm improves the TwigStack algorithm by introducing the *SJCursor*, which uses the XR-tree index to skip both ancestor and descendant data. However, it uses the same output model as the TwigStack algorithm, which generates output for every query node in a query pattern and merges all root-to-leaf paths into tree patterns in the end. This output method limited the opportunities for further optimization. Based on this observation, we will propose a simple output model that can enable us to output only necessary results and to skip more non-matching data.

5.3 The CB-tree Index

Before we go into the details of the CB-tree index, let us first look at why we need such an index.

5.3.1 Motivation

The numbering scheme discussed in Chapter 2 has been used in sort-merge based algorithms introduced in Chapter 4. One of the requirements is that both ancestor and descendant sets should be in sorted order by element ID, and be accessed sequentially. However, in the following situations, a sequential scan may not be the best choice. First, when we have two unevenly sized input sets, scanning the larger input sequentially may be costly and unnecessary. Let us use the Q1 in Section 4.1 again to illustrate the idea.

Q1: //book//section[figure][table]/title

In Q1 we may have a small number of `figure` descendants and need to verify if each of these descendants has a `section` ancestor. If we have a lot of `section` elements in the XML data, it is undesirable to scan all of them. Second, when the join operation of `section` and `figure` has a high selectivity, accessing the whole input data to produce only a few results is costly. Indexing is an effective technique that can be used in the above situations. With appropriate indexes, a query processor can quickly locate the data needed during a query processing. Before introducing the index structure proposed in this section, we shall first describe the operations that should be supported by an index.

5.3.2 Containment and Reverse Containment Queries

Consider a simple query `section//figure`. If there are a few `section` elements and a large number of `figure` elements, an index on `figure` elements can help to quickly find out if a `section` element has any `figure` descendants. In this dissertation, we refer to such queries as *containment queries*. A B⁺-tree index can be built on `figure` elements to support such an operation [15]. As mentioned before, each element is assigned a range according to the numbering scheme. Any `figure` element with a range contained in the range of a `section` element will be a descendant of the `section`. We can use the left position of the `figure` element range, which is also the element ID, as the key to build a B⁺-tree index. To verify if a `section` element has any `figure` descendants, we can use the starting position of the `section` element as the search key to search the `figure` index to find the first `figure` element larger than the key, and verify if the `figure` element is a descendant by checking if it is in the range of the `section`.

On the other hand, we could have only a few `figure` elements and a large number of `section` elements. To verify whether or not a `figure` has any `section` ancestors is the same as finding all the ranges that contain a given point. The B⁺-tree index

is not adequate to solve this problem. The reason is that the B⁺-tree is built using the element IDs. The ranges containing the given point can be anywhere in the B⁺-tree. The Segment Tree [6] and the Interval Tree [26] are data structures that have been developed in the field of Computational Geometry for indexing interval data. They are useful for detecting the intervals that contain a given point. Since ancestor ranges can be considered as line segments, we could use these structures to index ancestor ranges such as `section` elements and to get all `section` elements that contain a `figure` element quickly. In this dissertation, a query searching all ancestors having the same specified tag name for a given descendant is called a *reverse containment query*. Since the segment tree and the interval tree are in-memory data structures, they are not suitable for secondary storage. Another choice is to use the one dimensional R-tree to store the element ranges. The downside is that, during the search of an R-tree, there could be multiple search branches. Furthermore, each internal entry stores more information than a B⁺-tree, which makes the fan-out of the R-tree smaller. The search performance is not comparable to that of the B⁺-tree.

In the above examples, the `figure` element is considered as a descendant. It could also be an ancestor of other elements. In order to support both containment and reverse containment queries on the `figure` element, we could use two different index structures. For example, we could use a B⁺-tree to index `figure` elements for containment queries and use an R-tree to index the same set of `figure` elements for reverse containment queries. However, the storage and maintenance costs of the indexes are high. It would be ideal if we could combine the indexes for both containment and reverse containment queries into one.

The XR-tree [40] is an index that supports both containment and reverse containment queries. It stores copies of element ranges in stab lists in internal index nodes. In the next section, we propose another index structure, the CB-tree index. Unlike the XR-tree, the CB-tree stores replicated ranges in leaf nodes based on the observation that the index occupancy rate is not 100% and the replication is limited for

most XML data with a small number of element recursions. Since the index internal structure remains the same as the B⁺-tree, the implementation of CB-tree is simpler than that of the XR-tree.

5.3.3 The Containment B⁺-tree (CB-tree) Index

For simplicity, we describe the index structures and algorithms limited to a single document. It is not difficult to extend them to multiple documents. We also assume that an index is built on the elements with the same tag name.

If an element definition in an XML DTD or schema is not recursively defined, the ranges of the element instances with the same tag name are mutually exclusive. In this case, a B⁺-tree can be used to support both containment and reverse containment queries. If an element can recursively contain elements with the same tag name, then there can be overlaps among the corresponding ranges of these element instances. As mentioned in Section 5.3.2, the B⁺-tree is not adequate to support reverse containment queries in this case. To solve this problem, we propose the use of the Containment B⁺-tree (CB-tree) index, which is a simple but elegant extension of the B⁺-tree. The idea is based on replicating the elements whose ranges overlap with other ranges in different leaf pages. The internal structure of the CB-tree remains the same as the B⁺-tree, which uses the left value of each range as the key. Because of the replication, the entries in a leaf page of a CB-tree are classified into two categories, *normal entries* and *extra (replicated) entries*. The normal entries are the same as leaf entries in a traditional B⁺-tree. The extra entries in a leaf page are copies of the ranges (from other leaf pages) that cover at least one normal element entry in this leaf page.

In XML documents the number of recursively defined element tags is usually very small, as has been shown by Mignet *et al.* [53]. 99% of the documents with recursive content use only a few of different element tags. Most documents do not

have any recursively defined elements. They also showed that the recursive fan-out is distributed following a power law of degree 2.4, which means only very few elements have a large number of descendants. So, the number of extra entries in each leaf page is usually very small, which was also observed in our experiments. This provides the rationale to use *replication* in our indexes. Further evidence and explanation is provided in Section 5.5.3. We need to note that the worst case recursion would be an XML tree with a single path composed of elements with the same tag. If the length of this single path is too long ¹ (e.g., thousands of nodes), the CB-tree will suffer from the large amount of replication. For this kind of unlikely case, we would not recommend the CB-tree. From our investigation in Section 5.5.3, we will show that the depth of recursion is shallow for existing XML data sets resulting in an acceptable replication cost.

5.3.4 Storing Extra Entries in the CB-tree

The extra replicated entries can be stored in a leaf page or in a separate extra page pointed to by a leaf page. The following two methods can be used.

Closed In this method, the normal entries and extra entries are treated in the same way. There are no extra pages. The extra entries are always stored in leaf pages. The extra entries can affect the split and merge during insertion and deletion.

Hybrid Usually the index occupancy rate is not 100%. There is empty space in leaf pages. If the empty space is enough for the extra entries, then they are stored in the leaf pages directly. Otherwise, the extra entries are extracted out and stored in separate extra pages.

The illustrations of the above two methods are shown in Figure 5.2 and 5.3. The left and right position values of indexed ranges are shown in the entries of these two

¹Usually, an index page can hold hundreds of records.

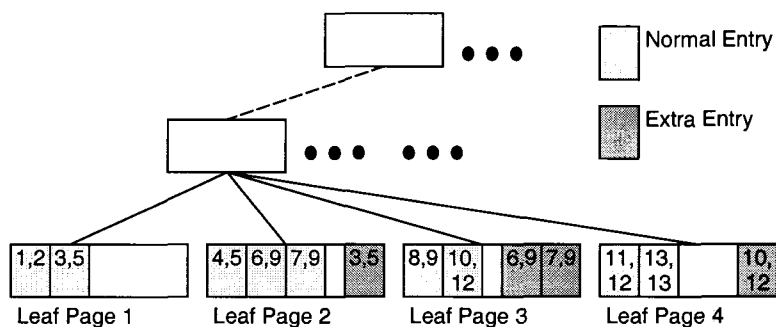


FIGURE 5.2. Closed Method to Store Extra Entries

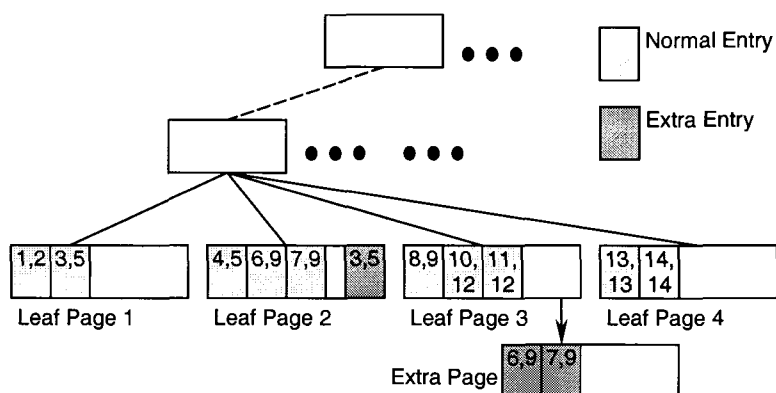


FIGURE 5.3. Hybrid Method to Store Extra Entries

figures. Note that the left position is the index key. The right position is also stored in the index such that containment relationship can be checked without having to access data. By storing the extra entries in separate extra pages, the disk utilization is reduced and the I/O cost increases. The hybrid method tries to put extra entries inside leaf pages and makes full use of the capacity of leaf pages. Since the number of extra entries is very small, this technique is quite effective. It eliminates almost all the extra pages, which is clearly shown in our experiments in Section 5.5.3.

Using the closed method, the insertion/deletion of an extra entry can trigger a split/merge operation. This makes the insertion and deletion more complicated than that in the hybrid method. The extra entries of the closed method affect the fan-out of the CB-tree, but there is no extra page search cost. The hybrid method, on the other

hand, retains the internal node structure of a B⁺-tree, but it may introduce extra page searches. Our experiments show that the performance of the closed method is almost the same as that of the hybrid method. Thus we recommend the use of the hybrid method, which needs very little modification of the existing B⁺-tree.

5.3.5 Operations on the CB-tree

When a new element range is inserted into the CB-tree, the range is inserted using its left position (element ID) as the key. It is first inserted as a normal entry similar to a traditional B⁺-tree insertion. Then this range is inserted as an extra entry to the leaf pages that overlap with this range. This can be done by following the next-page links on the leaf pages. If a split is needed at the leaf level, the extra entries will be re-scanned and some of them will be copied to the new leaf page. The internal page split remains the same as in a B⁺-tree.

When an element range is deleted from the index, the normal entry will be deleted from the leaf page. Then, a scan is needed to remove all replication of the range if there is any. If the leaf page occupancy is below the threshold, two leaf pages can be merged together. Some of the extra entries may be removed during this merge because they may result in duplicate copies in the merged leaf page.

For containment queries, the search using the CB-tree is just like a range search in a traditional B⁺-tree. No extra entries are accessed in this search. For example, if we need to find the descendants of range [3,5], we take 3 as the key to search the CB-tree, which will lead us to the first page in Figure 5.2 or Figure 5.3. Then we scan the leaf pages to find all records with keys less than 5. To answer the reverse containment query, the index search of the CB-tree can be divided into two steps. The first step is to search the normal entries and the second step is to search the extra entries. The operation and the cost to search for normal entries are the same as that of a B⁺-tree. Note that the extra entries can be stored in the same leaf page. In

Field	Description	Initial value
<code>kind</code>	Query node kind in output model	<code>critical,</code> <code>target</code> or <code>normal</code>
<code>cursor</code>	Provides access to elements in the associated index	index cursor
<code>cur_range</code>	The most recent local answer found so far	-1
<code>cache</code>	The set of ranges containing the largest child's <code>cur_range</code>	\emptyset
<code>res_range</code>	Result range for child query nodes to filter results	<code>[-1, -1]</code>
<code>result</code>	The result set for the query node	\emptyset

TABLE 5.1. The Structure of Query Nodes

this case, there is no extra I/O cost for the extra entry search. Otherwise, the extra pages will be accessed. We shall use Figure 5.3 as an example to illustrate the reverse containment query. Suppose we are going to search for all the ranges that cover key 8. First, we locate the key 8 in leaf page 3 using the same search method as B⁺-tree. From the normal entries, we get the range [8, 9]. Then following the extra page, we get ranges [6, 9] and [7, 9] that cover key 8. We can see that the search is limited on one leaf page and its associated extra pages, if any.

5.4 The IndexTwig Algorithm

In this section, we introduce an index based twig join algorithm, *IndexTwig*, which accepts a twig query and utilizes the CB-tree index for efficient evaluation. For a twig query, intermediate results could be large if we decompose the query and process each join separately. Following this observation, the IndexTwig algorithm processes a twig query as a whole and avoids breaking up the twig and generating intermediate results.

In the following, we describe the IndexTwig algorithm assuming all data belongs to the same document. The extension to multiple documents is straightforward.

5.4.1 The Structure of Twig Query Nodes

We shall refer to a node in a query twig as a *query node*. The fields associated with a query node are shown in Table 5.1. Note that since each element is assigned a unique range according to the numbering scheme, we use ranges to refer to elements. The `kind` field is used to differentiate different output roles in the output model, which will be introduced in Section 5.4.6. Each query node has a `cursor` for accessing the associated index, which indexes all elements having the same tag name as the query node by their ranges. The cursor retains the position of the most recent access. The `next` operation on a cursor returns the next element in the index without moving the cursor, while the `advance` operation moves the cursor forward by one element without returning it. The cursor supports the `jumpTo` operation, which directly jumps to the specified location. The `findAncestors` operation on a cursor returns a set of ancestor ranges, each of which covers the cursor. The returned ranges are stored in the `cache` of query nodes, which will be detailed in Section 5.4.4. The `result` field is used to hold the answers for a query node. We will describe the current range `cur_range` and the result range `res_range` in the following sections.

5.4.2 Basic Idea of the Algorithm

In this section, we describe how non-matching data are skipped using indexes. First we introduce the data skip of parent-child pairwise patterns. Then the idea is extended to parent-children one-to-many simple twig patterns, and further to arbitrary twig patterns. Note that, this parent-child or parent-children relationship for query nodes refers to relationships inside a query pattern, not in XML data tree. For example, the path query, `book//section`, finds `book` and `section` elements in ancestor-descendant relationship. On the other hand, this query is a parent-child query pattern, since `book` query node is the parent of the `section` query node in the query node tree.

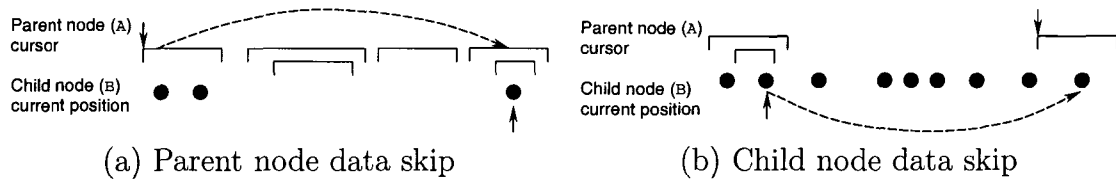


FIGURE 5.4. Parent-Child Data Skips

Parent-Child Skip Using sort-merge based algorithms to process a pairwise pattern such as $A//B$, the two sorted input element lists, for A and B elements, are scanned sequentially. With indexes, we can skip elements in these lists to avoid non-matching data access. This idea is illustrated in Figure 5.4, where the elements of query node A are shown as a range list, and the descendants of query node B are shown as a point list. The arrow of descendant list is the position of `cur_range`, which is the largest local answer found so far. The *local answer* is a range that satisfies a sub-tree of a twig. In the parent-child case, a local answer for A would be a range that covers at least one point. The arrow of ancestor list is the same as the cursor position. In Figure 5.4(a), the child `cur_range` is ahead of the parent cursor. With indexes, the parent cursor can directly jump to the position of child `cur_range` by performing a *reverse containment* search using the child position as the key. Non-matching ranges are skipped without missing those that should participate in the final answer, since the reverse containment search returns all possible ranges covering the child `cur_range` position. In Figure 5.4(b), the child `cur_range` can jump to the parent cursor position using a B^+ -tree lookup, which is also supported by the CB-tree. Thus unnecessary data in both the ancestor and descendant lists can be skipped.

Parent-Children Skip We can extend the same idea to a query pattern with one parent query node and multiple children query nodes. We shall call this query pattern *simple twig*. For example, the query, `section[figure][table]/title`, is a simple twig query with one parent query node `section` and three children query nodes, `figure`, `table`, and `title`. The illustration of how the data are skipped is shown in Figure 5.5.

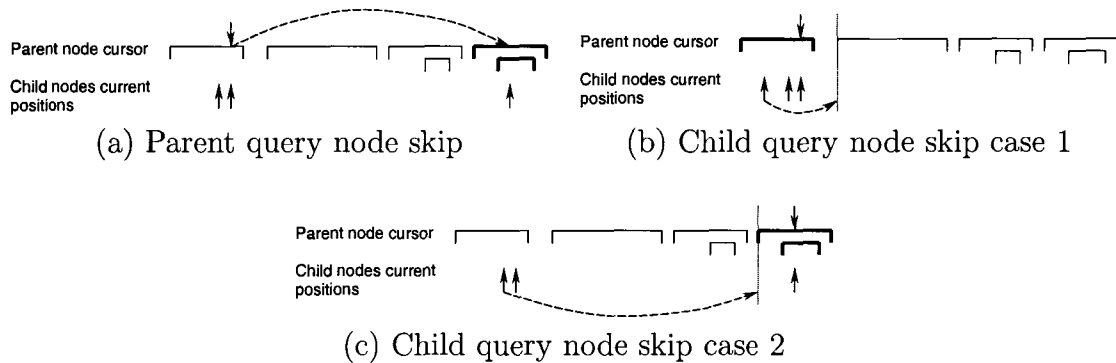


FIGURE 5.5. Parent-Children Data Skips

There are three child `cur_range` positions in this figure. The parent query node skip in Figure 5.5(a) is the same as that in Figure 5.4(a), except that the parent cursor jumps to the *largest child*, the child with the largest `cur_range`. The reason is that any answer range should cover all three positions and hence it should at least cover the largest one.

The child query node data skips can be divided into two cases. Figure 5.5(b) shows the first case when all child current positions are contained by the same set of ranges. In this case, all child current positions can jump to the start point of the next range in parent cursor, which is marked with a vertical line. Figure 5.5(c) shows the second case when one child is ahead of others. Other children can jump to the range that might be the next answer, which is marked with a vertical line in the figure. Other children also need to jump there to verify if the possible answer is a real local answer. In this way, both child and parent query nodes can *jump* forward to skip the data that do not participate in query processing.

It is not difficult to see that any complex twig pattern can be constructed using simple twigs. For example, Figure 5.6 illustrates that Q2 is composed of four simple twigs. Thus, the idea of skipping data in simple twigs can be generalized to any twig pattern. The IndexTwig algorithm described in the following sections implements this idea. It recursively treats each node as a simple twig to skip data. During this

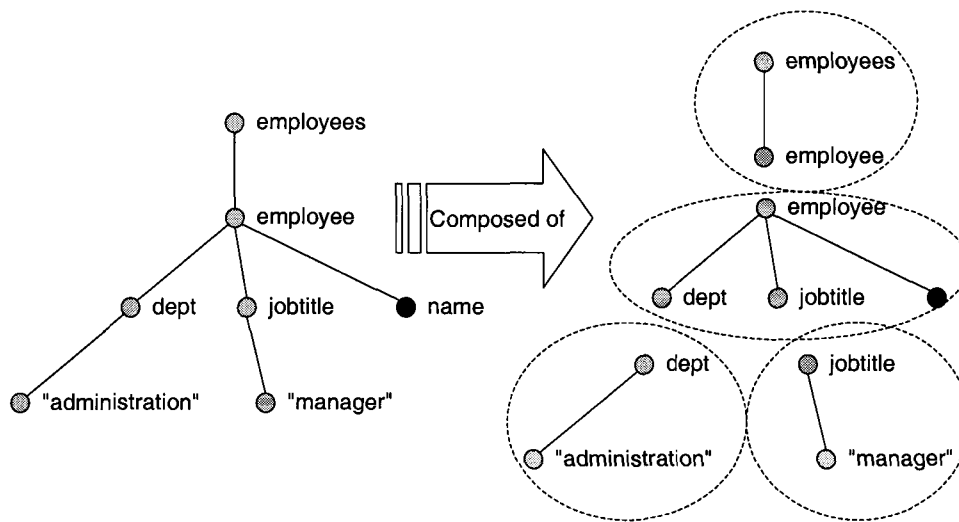


FIGURE 5.6. A Complex Twig and Its Simple Twigs

recursive processing, one important benefit is that the jump positions of ancestor nodes can be passed down to descendant query nodes to help improve the jump further.

5.4.3 Outputting Answers

As we have described before, we call a range a *local answer* when it satisfies a sub-tree of a twig. Accordingly, a range that belongs to a match of a whole twig is called a *global answer*. It is possible that a local answer may not be a global answer. For example, in Q1 any `title` element is a local answer, since it satisfies the leaf query node `title`. However, not all `title` elements are global answers. We must identify such non-global answers and exclude them from being output. To do this, a result range, `res_range`, is associated with each query node. The result range itself is a global answer and it is used to determine whether a local answer of its child query nodes is a global answer. Since the root query node does not have a parent, the result range for the root to filter local answers is the universal range. The result range is updated when a new global answer is found. The algorithm to output results and

Algorithm 6: OutputAnswer: Output answer ranges and update the result range if possible.

```

input :  $Q$  — query node;
          $S$  — the set of possible answer ranges;
          $pr$  — parent result range;

procedure OutputAnswer( $Q, S, pr$ )
1: foreach  $r \in S$  do
2:   if  $pr$  contains  $r$  then
3:      $Q.result = Q.result \cup \{r\}$ ;
4:     if  $r.left > Q.res\_range.right$  then
         /*  $r$  should be the new result window */
5:        $Q.res\_range = r$ ;
         end
       end
     end
  end

```

update result ranges is shown in Algorithm 6. If a new answer of Q is out of the result range of itself, the new answer will be used as the new result range for Q (Line 4 to Line 5). The reason is that the new answer range covers all child nodes' current ranges, since it is an answer. And all future child local answers to be output will be greater than the new answer.

5.4.4 The Range Cache of a Query Node

During a twig query processing, the parent cursor jumps along with the largest child current range position. When a child `cur_range` jumps to a new position and becomes the largest child, the parent cursor is moved to that position. At the same time we obtain all ranges containing the largest child `cur_range`. For example, in Figure 5.5(a) there are two ranges containing the new child current position. With the support of indexes, we can retrieve these ranges using the reverse containment query. These ranges returned are in sorted order and are stored in a range cache of a query node. The cached ranges in Figure 5.5 are highlighted in thick lines. The cache is always maintained according to the largest child `cur_range` position. Only these ranges in cache are possible to cover all child current ranges and are possible answers.

5.4.5 Output Model

Existing twig join algorithms, such as TwigStack and TSGeneric+, output all matching patterns of a twig query. In order to do that, every root-to-leaf path of a matching pattern is generated. The generated paths are merged together in the end. In XQuery [7], path expressions are used to locate nodes within a document tree. The final result of a path expression is the *node sequence* that results from the last step in the path. For example, the result of path expression Q1 is a sequence of `title` elements. The number of matching patterns of Q1 could be much larger than the size of resulting `title` sequence, since one section could contain many figures and tables. Hence, to find all patterns and output them is potentially costly. It is also unnecessary to output all the matching patterns for a path expression, because the result that we need is a sequence of elements. Based on this, the IndexTwig algorithm only outputs elements for the target query node instead of the whole matched patterns.

5.4.6 Fast Existence Test

The output model of the IndexTwig algorithm also provides the opportunity to further optimize a twig join algorithm. We shall use Q1 as the example to explain the optimization idea. In Q1, the `figure` and `table` elements are used as existential predicates to filter the `section` element. For an existential predicate, it is enough to know the result is not empty. We do not have to determine all the results. For example, a `section` element may have many `figure` elements as children. When we find one `figure` element, we immediately know the result of the predicate, `[figure]`. The rest of the `figure` elements in the same section are not useful anymore. Thus we can safely skip them. In this dissertation, we shall refer to the technique of evaluating the existential predicates such as `[figure]` without having to examine every instance of that predicate (and avoid generating unnecessary answers) as *Fast Existence Test* (FET) optimization. The FET optimization helps in skipping unnecessary

Algorithm 7: GetJumpPos: Return the Jump Position

```

input :  $Q$  — parent query node
output: The Jump Position for Children

function GetJumpPos( $Q$ )
1:  $Q' =$  the smallest child of  $Q$ ;
2: if  $Q.cache.end() < Q' \vee Q.cache = \emptyset$  then
3:    $pos = Q.cursor.next.left$ ;
   else
4:    $r =$  the first range in  $Q.cache$  greater than  $Q'$ ;
5:    $pos = r.left$ ;
   end
6: return  $pos$ ;

```

data during the existence test and omitting unnecessary answers.

Before we describe the FET implementation, let us first define the target node and the critical nodes in a query pattern. A *target node* is the node that a query processor should generate answers for. For example, `title` is the target node in Q1. All the ancestors of the target node in a query pattern are called *critical nodes*. For example, in query Q1, `section` is a critical node. If a query node is a target node, then every global answer of this node should be output to the result set. For critical nodes, every global answer should be examined, but it is not necessary to output them. For other query nodes, some answers may be skipped since we are not interested in finding out all of them.

The above idea can be easily applied to the *IndexTwig* join algorithm described in Section 5.4.8. We use `kind` in the query node data structure to distinguish different kinds of nodes in the algorithm. The value of `kind` can be `critical`, `target` or `normal`.

5.4.7 Determining Jump Positions for Child Query Nodes

Where a parent query node should jump to can be found easily. It is the same as the `cur_range` of the largest child. For a child query node, we need to find out the new

position for its `cur_range` to jump. The algorithm to do this is shown in Algorithm 7. Basically, the algorithm follows the two cases in Figure 5.5. In the algorithm, Line 2 and 3 correspond to the first case of child skip in Figure 5.5(b). In this case, the largest range in the `cache` of the parent query node is smaller² than the smallest child (or the `cache` is empty), which means either there are no ranges in the `cache` or these cached ranges have been processed already. The next possible answer range can only be the next range in the parent cursor.

When there is a range in the `cache` that is greater than the smallest child, we need to verify if that range is an answer. So, it is the position for a child to jump. This is realized by Line 4 and 5 in Algorithm 7. This case also corresponds to the case in Figure 5.5(c).

5.4.8 The IndexTwig Algorithm

The main body of the algorithm is shown in Algorithm 4. The input query pattern is stored as a tree structure pointed to by a root query node *root*. The `IndexTwig` algorithm repeatedly invokes the `HasAnswer` function with the minimum jump position *min* and the result range `[0, INT_MAX]`. Note that the `cur_range` field of *root* will be changed by the `HasAnswer` function when a new answer is found. The `HasAnswer` function shown in Algorithm 9 takes three parameters, a query node *Q*, a minimum jump position *min*, and a parent result range *pr*. The function returns `TRUE` if it finds a local answer larger than the parameter *min*. Before return, the local answer is stored in the `cur_range` of the query node *Q*. In the `IndexTwig` function, the minimum jump position *min* is updated to the newly returned answer, such that the next answer to be returned by `HasAnswer` will be greater than this one (Line 3, Algorithm 4). In this way, the results are retrieved one by one in increasing order.

At the beginning of the `HasAnswer` function (Algorithm 9), the current range

²When ranges are used in comparison, the left positions of ranges are used.

Algorithm 8: IndexTwig: The main function.

input : *root* — the root node of query tree

procedure *IndexTwig*(*root*)

1: *min* = *root.stream.next* - 1;

2: **while** *HasAnswer*(*root*, *min*, [0, INT_MAX]) **do**

3: *min* = *root.cur_range*;

end

4: *FlushAnswer*(*root*, [0, INT_MAX]);

cur_range, which is the local answer found in the previous invocation of *HasAnswer*, is output. If the query node is a leaf (Line 2), the answer ranges less than *min* will be output only if the query node is the *target* node (Line 5). Otherwise, they can be skipped, which is one benefit of the output model and the FET optimization. Then the search simply jumps to *min* and returns the next range in the *cursor* (Line 6 to 9), since any range greater than *min* is a local answer.

If the query node is not a leaf node, then the *HasAnswer* function enters a **while** loop (Line 10). At the beginning of the loop (Line 11), the termination condition is tested. If no more results exist, *FALSE* will be returned.

The *IsAnswer* function, which is not shown, is a simple function to test if a range covers all child current ranges and to determine if a range is a local answer. The local answers in the range cache can be divided into two sets. The ranges in set *A1* are less than *min*. These answers are of no use for parent query node of *Q*. So, the algorithm just outputs them in Line 14. The other set *A2* contains answers greater than *min*. If *A2* is not empty, the smallest range will be the current range of *Q* (Line 17). The *TRUE* will be returned since we found a local answer greater than *min*.

If *A2* is empty, we need to advance the current ranges of child query nodes and use the advanced current ranges to new find local answers larger than *min*. To do this, we need to find the minimum position for the children to advance by invoking the *GetJumpPos* function in Algorithm 7 Line 19. If the query node *Q* is a normal node, we do not need to find all the local answers less than *min*. So, we can let

Algorithm 9: HasAnswer: Find local answer.

```

input :  $Q$  — query node;
          $min$  — the minimum answer;
          $pr$  — parent result range;
function  $HasAnswer(Q, min, pr)$ 
1:  $OutputAnswer(Q, \{Q.cur\_range\}, pr)$ ;
2: if  $Q$  is a leaf query node then
3:    $A0 = \{a \in Q.cursor \mid Q.cur\_range < a \leq min\}$ ;
4:   if  $Q.kind = target$  then
5:      $OutputAnswer(Q, A0, pr)$ ;
6:   end
7:    $Q.cursor.jumpTo(min)$ ;
8:   if  $Q.cursor$  is empty then return FALSE;
9:    $Q.cur\_range = Q.cursor.next$ ;
10:  return TRUE;
else
11:  while TRUE do
12:    if  $Q.cache$  and  $Q.cursor$  are empty then
13:      return FALSE;
14:     $A1 = \{a \in Q.cache \mid IsAnswer(a, Q) \wedge$ 
15:       $Q.cur\_range < a \leq min\}$ ;
16:     $OutputAnswer(Q, A1, pr)$ ;
17:     $A2 = \{a \in Q.cache \mid IsAnswer(a, Q) \wedge a > min\}$ ;
18:    if  $A2 \neq \emptyset$  then
19:       $Q.cur\_range = Minimum(A2)$ ;
20:      return TRUE;
21:    end
22:     $c\_min = GetJumpPos(Q)$ ;
23:    if  $c\_min > pr.right \vee Q.kind = normal$  then
24:       $c\_min = Maximum(c\_min, min)$ ;
25:       $Q' = PickChild(Q)$ ;
26:      if not  $HasAnswer(Q', c\_min, Q.res\_range)$  then return FALSE;
27:      if  $Q'$  becomes the largest child then
28:         $Q.cache = Q.findAncestors(Q'.cur\_range)$ ;
29:      end
30:    end
31:  end

```

Algorithm 10: FlushAnswer: Flush answers for target node in the end.

```

input :  $Q$  — a query node;
          $pr$  — parent result range;
procedure FlushAnswer( $Q$ ,  $pr$ )
1: for  $Q'$  in  $Q$ .children  $\wedge$   $Q'$ .kind  $\neq$  normal do
2:   repeat
3:      $min = Q'.cur\_range.left$ ;
4:      $found = HasAnswer(Q', min, pr)$ ;
       until  $not\ found \vee Q'.cur\_range > pr$ ;
5:     FlushAnswer( $Q'$ ,  $Q.res\_range$ );
end

```

children jump to min if min is larger (Line 21). We can also make c_min larger if c_min is larger than the right end of parent result range $pr.right$ (Line 20). Since the parent query node of Q does not have answers between $pr.right$ and min , Q will not have answers either. The larger c_min value the more data we can skip. It is another benefit of using the output model and the FET optimization.

The `PickChild` function determines which child will be selected to advance. In our current implementation, the `PickChild` function directly returns the smallest child. Sophisticated optimization can be applied here to return the child that can advance most and skip more data. After the recursive call of `HasAnswer` (Line 23), if there are no more local answers for the child, `FALSE` is returned. Otherwise, we check if the new current range of Q' becomes the largest position. If so, a new set of range cache is retrieved by the `findAncestor` function. Thus, $Q.cache$ is maintained according to the *largest child*, whose current range `cur_range` is the largest among all the children.

When the while loop (Line 10) terminates, either there are no more local answers, or a local answer range larger than min has been found. If a new local answer is found, it is stored in `cur_range` (Line 17). This range will be used by the parent query node of Q to determine the parent's own local answers.

When the while loop in `IndexTwig` algorithm (Algorithm 4) terminates, all an-

swers of root has been found. At this time, it is possible that answers for other query nodes may have not been output. For example, in query Q1, when the last **section** answer is found, some **figure** answers contained in this section may have not been visited at all. The **FlushAnswer** function in Algorithm 10 will exhaustively search each query node for answers within the existing result ranges. Thus, we do not miss global answers for descendant query nodes. Also, according to the output model, the normal query nodes do not need to be flushed (Algorithm 10 Line 1). This optimization skipped the unnecessary data access.

5.4.9 Index Independency

From the description of the IndexTwig algorithm, we can see that the algorithm does not depend on any particular index structure as long as the index can support the containment and reverse containment queries. This flexibility makes it possible to use other indexes besides the CB-tree.

5.4.10 Algorithm Illustration

We use a path expression, `/book[.//section[table][figure]]`, to illustrate the query processing of the IndexTwig algorithm. This path query finds all the **book** elements with a **section** element that contains both **table** and **figure** elements. The processing illustration is shown in Figure 5.7. In each figure, we show the query pattern on the left side. The element lists of internal query nodes (**book** and **section**) are represented using range lists. The element lists of leaf query nodes (**table** and **figure**) are represented using point lists.

At the beginning of the processing, in Figure 5.7(a), the cursors are initialized pointing at the heads of the lists. The algorithm first processes the **book** query node and asks the **section** node to return any answers greater than the start position of the **book** node. The **section** query node in turn asks its children for answers. When the

`table` node returns its local answer, the cursor of the `section` node jumps forward and get a set of candidate answers (in dashed lines) using a reverse containment query, which is shown in Figure 5.7(b). In Figure 5.7(c), after the `figure` query node also returns its local answer, the `section` node now can make sure the candidates are its local answers (in thick lines) and returns them back to the `book` element, which finds the first global answer. Then the `book` element advances its cursor to the next element (Figure 5.7(d)). Again, the `section` node gets a set of candidates in Figure 5.7(e). But, the `figure` node returns a local answer that is far ahead (Figure 5.7(f)). So, the `section` node retrieves another set of candidates using a reverse containment query. After verifying the candidate by the answers returned from the `table` node, the `section` node returns this local answer back to the `book` node, which finds another global answer. This is shown in Figure 5.7(g). In Figure 5.7(h), the `book` node advanced its cursor to the next range. This position is passed down to the `section` node, which in turn passes it to the `table` node. But the `table` node has no more elements in its list. The algorithm then halts the processing and flushes global answers if there is any.

From this illustration, we can see that the jump positions can be passed from ancestor query nodes down to their descendant query nodes. Also, the jump positions of descendant query nodes can be passed back to ancestor query nodes. In this way, the cursors of query nodes jump forward and skip non-matching data efficiently.

5.5 Performance Study

In this section, we present the performance study of the CB-tree indexes and the IndexTwig algorithm. We implemented the TwigStack [10], TwigStackXB [10] and TSGeneric+ [41] algorithms for our comparison. From our preliminary experiments, we found out that the performance of TwigStackXB in general was slower than TSGeneric+ and IndexTwig, which conformed to the result claimed by Jiang *et al.* [41].

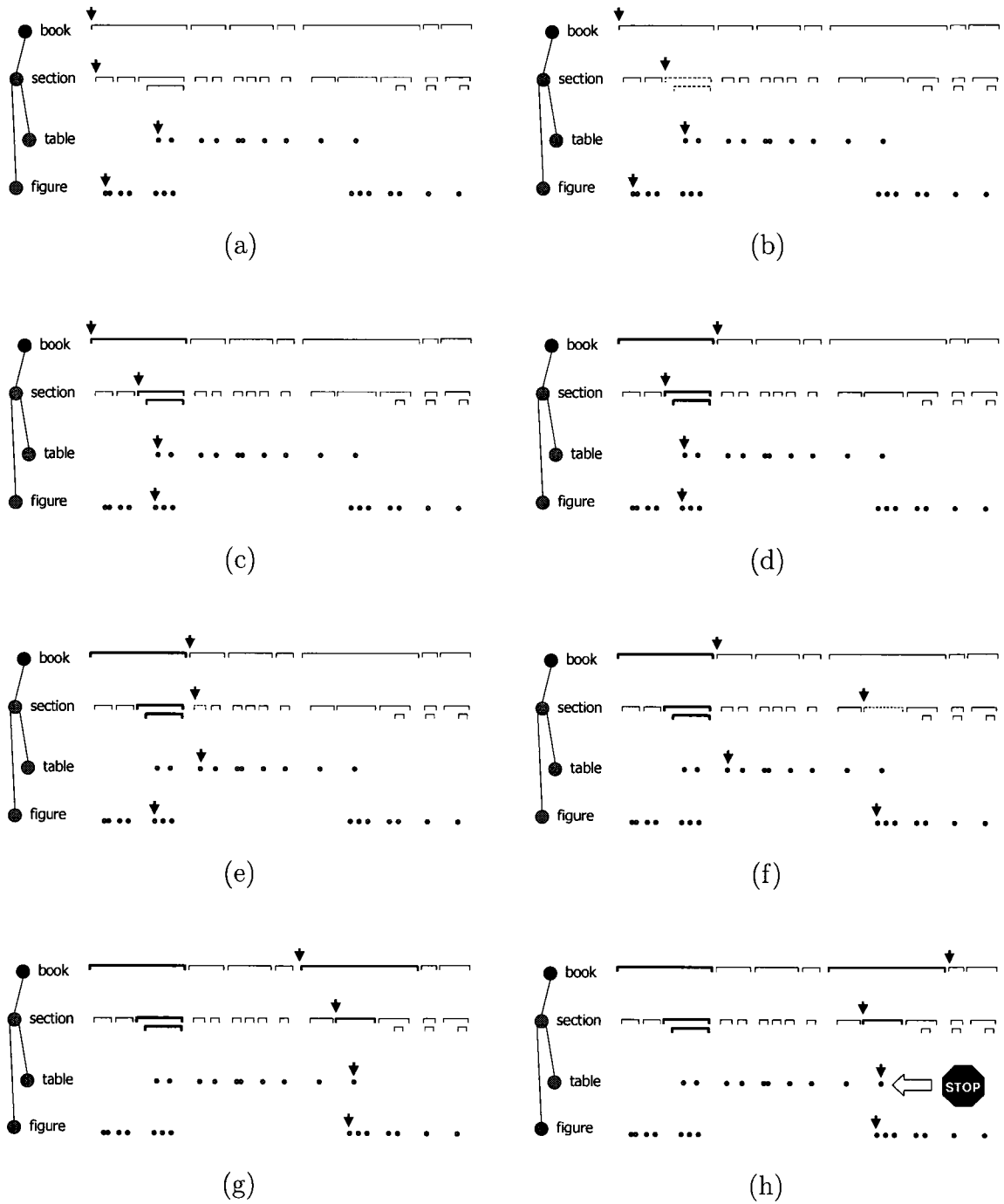


FIGURE 5.7. Algorithm Illustration

Query	TwigStack	TwigStackXB	IndexTwig
DBLP1	7964	700	372
DBLP2	9322	2402	402
NITF1	1685	1196	102
NITF2	1477	581	73
Shaks1	1366	1366	140
Shaks2	516	40	18

TABLE 5.2. Number of I/O Operations: TwigStack, TwigStackXB and IndexTwig

For example, Table 5.2 shows the I/O performance of TwigStack, TwigStackXB and IndexTwig for the first six queries in Section 5.5.2 Table 5.3. The IndexTwig significantly outperformed TwigStack and TwigStackXB. Therefore, in the rest of this section we only present the performance comparison between the IndexTwig and the TSGeneric+ algorithms.

For indexes, we implemented the XR-tree [40] and the CB-tree on top of a paged file layer, which also provided the memory buffer management functionality. In our experiments, the page size was 4K bytes. Each entry in a leaf page consisted of four 4 bytes integers namely document ID, element ID, size (the size of descendant), and the level of the element in the document tree. For the CB-tree, two versions (Closed and Hybrid) were implemented as described in Section 5.3.

Experiments were performed on an Intel workstation with a Pentium 4 1.6GHz CPU running Solaris 8 for Intel platform. This workstation had 512M bytes of memory and a 40GB EIDE disk drive (with 7200 RPM and 8ms average seek time). The disk was locally attached to the workstation and was used to store XML data. We used the direct I/O feature of Solaris for all our experiments to avoid operating system's cache effects.

5.5.1 Data Sets and Performance Metrics

We experimented with four data sets, Shakespeare, DBLP, TREEBANK and XMark, from real-world applications and XML benchmarks. We also used one synthetic dataset generated by the XML Generator [24]. We have mentioned the Shakespeare and DBLP datasets before. Below is a brief description of the rest datasets we used.

NITF: This dataset was generated using the XML Generator [24] with the DTD of the News Industry Text Format (NITF) [38]. The dataset had 225 MB with 100 separate document files of various sizes. Note that this dataset is different from the previously used NITF1 and NITF100 datasets.

TREEBANK: This dataset (82 MB) was obtained from the University of Washington XML repository [54]. The document trees in this dataset were skinny and had deep recursions.

XMark We used the 100 MB ready-made document in our experiments [58].

In our experiments, we measured the elapsed time (both CPU and IO) for query processing. For fair comparison, the output cost and the path merging cost in the TSGeneric+ algorithm were not counted.

5.5.2 Twig Queries for Performance Study

The XPath queries shown in Table 5.3 were used for performance evaluation. Most of these are complex twig pattern queries. These queries have different characteristics. Some of them are single long paths, e.g., TB3 and XM3. Some of them have bushy twig structures, e.g., DBLP2, NITF1 and Shaks1. There are value predicates for half of the queries, while there is no value involved for the remaining queries. From the result size table in Table 5.4, we can see that these queries also have various selectivity.

Query	Data Set	Path Expression
DBLP1	DBLP	//dblp[.//author="Dan Suciu"] [.//year="1998"]//title
DBLP2	DBLP	//dblp[.//year="1998"] [.//author] [.//cite]//title
NITF1	NITF	//body[.//block[.//note] [.//body.content] [descendant-or-self::text()="level 3"]
NITF2	NITF	//body[descendant-or-self::text()="level 3"] [.//block] [.//body.content]
Shaks1	Shakespeare	//PLAY[.//ACT[.//SPEECH[SPEAKER] [LINE]]]//TITLE
Shaks2	Shakespeare	//PLAY[.//ACT//SPEECH/SPEAKER="KING HENRY V"] //TITLE
TB1	TreeBank	//S//NP//SYM
TB2	TreeBank	//NP[.//RBR_OR_JJR]//PP
TB3	TreeBank	//NP//PP//NP[.//NNS_OR_NN] [.//NN]
XM1	XMark	/site/open_auctions/open_auction [bidder/date="06/04/2000"]
XM2	XMark	/site/regions//item[location="United States"]/name
XM3	XMark	/site/closed_auctions/closed_auction[annotation/ description//emph//keyword]

TABLE 5.3. Twig Queries for Performance Study

Query	# of results	Query	# of results
DBLP1	21431	TB1	7
DBLP2	19503	TB2	17
NITF1	19	TB3	8
NITF2	19	XM1	60
Shaks1	1031	XM2	16294
Shaks2	190	XM3	670

TABLE 5.4. Result Sizes of Twig Queries

We first analysis the CB-tree and the XR-tree and their replication costs. Then we compare the performance of IndexTwig and TSGeneric+ algorithms.

5.5.3 Index Replication Cost

As we have described in Section 5.3, if there is no recursive nesting in XML data, there is no replication in the CB-tree. Among the data sets we used, the Shakespeare data set has no recursive element. So, the CB-tree of this data set is the same as the traditional B⁺-tree.

In DBLP data set, most elements are not recursively defined except a few inside the definition of `title`. They are `sup`, `sub`, `i` and `tt`. These four elements are used to describe the typesetting of a `title` content. Although they are recursively defined, it is rare to see them deeply nested. So, if there is any replication, the number is very limited.

We also investigated several XML benchmarks including XMark, XMach-1³, and XBench⁴ to support our observation that the recursive nesting appears infrequently in most applications. In XMark DTD, there are recursively defined elements such as `listitem` and `emph` inside the `description` element. The recursive nesting of typesetting elements such as `emph` and `bold` is limited. Other recursively defined elements such as `listitem` and `parlist` have a recursive nesting level less than three. There was at most one extra entry on each leaf page after we loaded them into the CB-tree index. The element `section` in XMach-1 and the element `subsec` in XBench are recursively defined elements. They are used to describe the document structure. It is rare to see a document having more than five nested sections. So, the recursive nesting level is also limited in those two benchmarks.

We checked the Protein Information Resource(PIR) data set⁵ and several other

³<http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html>

⁴<http://db.uwaterloo.ca/~ddbms/projects/xbench/>

⁵<http://pir.georgetown.edu/>

Element Indexed	XR-tree			CB-tree (Hybrid)		
	Stab Element	Extra Pages	Total Pages	Extra Entries	Extra Pages	Total Pages
NITF block	492	5	543	532	1	548
NITF body.content	578	5	515	746	1	511
NITF note	73	1	137	82	1	139
DBLP author	0	0	2921	0	0	2911
DBLP title	0	0	1297	0	0	1284
Shakespeare LINE	0	0	572	0	0	567
Shakespeare SPEAKER	0	0	175	0	0	173
TREEBANK S	1021	9	947	985	1	944
TREEBANK NP	2154	17	2417	2108	6	2389
TREEBANK PP	349	8	841	315	6	832
TREEBANK NN	0	0	1036	0	0	1037
XMark date	0	0	518	0	0	514
XMark item	0	0	129	0	0	127
XMark name	0	0	265	0	0	263

TABLE 5.5. Index Information of the CB-tree and the XR-tree

data sets. We observed in most real applications that the number of recursively defined elements is small and the recursive nesting level of these elements is shallow. The replication cost of those applications would be very small. For these applications, the access cost of the CB-tree is almost the same as that of the B^+ -tree.

The small replication cost of the CB-tree was also demonstrated in our experiments. In Table 5.5, we gathered the index information of the typical elements used in our queries. To compare with the XR-tree, the corresponding XR-tree index information is also shown in Table 5.5. We proposed two methods to handle replicated entries, Hybrid and Closed. Since their index sizes are similar, we only show the Hybrid CB-tree in the table.

For data sets without recursive elements, e.g., the DBLP and Shakespeare datasets, there is no stab elements in the XR-tree or extra entries in the CB-tree. The XR-tree has a larger internal node key structure (the start and end positions of stab lists are stored), which affects the index fan-out and makes the index size a little bit larger.

For indexes with 4 KBytes page we used, the XR-tree is about one percent larger in size.

For datasets with recursive elements, e.g., NITF and TREEBANK, the index sizes of CB-tree and XR-tree are very close to each other. A stabbed element in the XR-tree has to be stored in a separated page. On the other hand, a extra element entry in the CB-tree can be stored in a leaf node if there is room. This is the reason why the XR-tree indexes of recursive elements, S, NP and PP, are larger than the corresponding CB-tree indexes.

As described in Section 5.3, if there is no recursive nesting in XML data, then there is no replication. In most real world applications, the number of recursively defined elements is small and the recursive nesting of these elements is shallow. The replication cost for these applications would be very small. However for highly skewed data with very deep recursive nesting, it is expected that the XR-tree would have less replication cost, because each element would be copied at most once.

5.5.4 Algorithm Performance

In this section, we compare the performance between IndexTwig and TSGeneric+. In the algorithm description of TSGeneric+ [41], the Algorithm 6 may enter an infinite loop if the query node q does not end (`not end(q)`) but a none-leaf cursor p reaches the end (`end(Cp)`). In our implementation, we let function `SJCursor` return a flag indicating if a cursor reaches the end. Algorithm 6 used this information to break the loop and return. If no extension found in Algorithm 5 (`getNextExt`) Line 4, return q may cause another infinite loop and incorrect semantics. Under this situation, we let the algorithm continue run on Line 5 instead of return. Also in the TSGeneric+ algorithm, we used the TD heuristic since we do not assume we have statistics.

Figure 5.8 and Figure 5.9 show the performance results in total elapsed time and I/O count (pages accessed from disk) to process the queries listed in Table 5.3. The

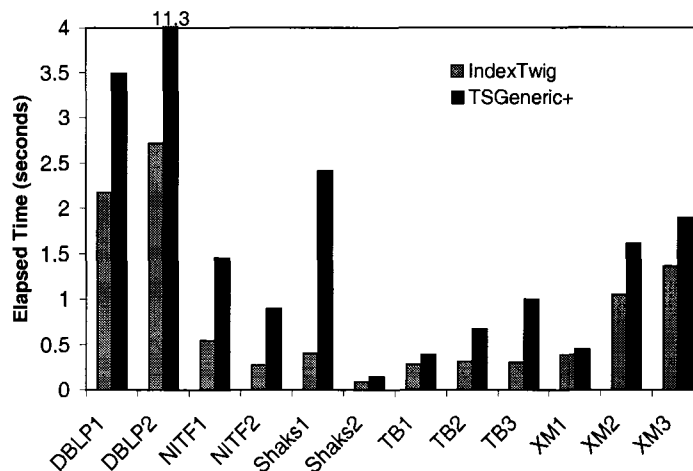


FIGURE 5.8. Elapsed Time IndexTwig vs. TSGeneric+

elapsed time was almost proportional to the I/O cost. IndexTwig performed better than TSGeneric+ for all twelve queries. Especially for query DBLP2 and Shaks1, IndexTwig is 5 to 9 times better than TSGeneric+.

In order to get rid of the effects caused by using different indexes, we implemented the IndexTwig algorithm on top of the XR-tree, since the XR-tree can support reverse containment query, therefore it can be used by IndexTwig. We found out that the performances of using different indexes were almost the same as shown in Figure 5.10. (The elapsed time had the same trend as the I/O performance. Only the I/O performance is shown in the figure.) This also conforms to the index comparison in Section 5.5.3. Since the index sizes of CB-tree and XR-tree were very close, using different indexes did not change the performance much.

After this comparison, we concluded that the performance gain of the IndexTwig algorithm over the TSGeneric+ algorithm can be attributed to algorithms rather than indexes. In TSGeneric+, the elements that were part of a query pattern match would be accessed and output. However, in many cases, these elements did not contribute to the correctness of the answers that should be output by the path expressions. According to our output model, only the answers of the *target* node should be picked

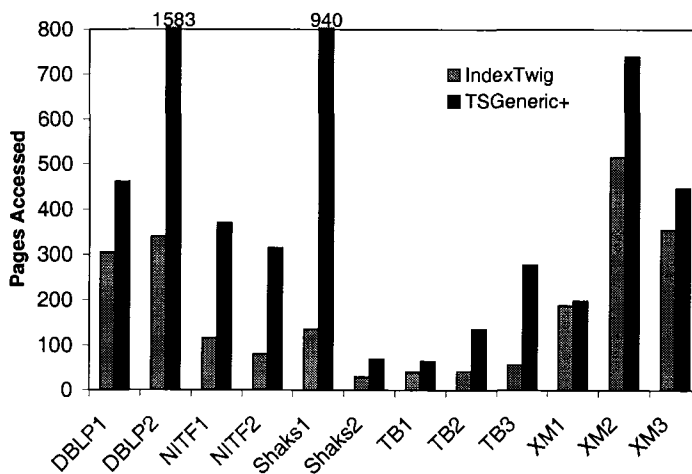


FIGURE 5.9. I/O Performance IndexTwig vs. TSGeneric+

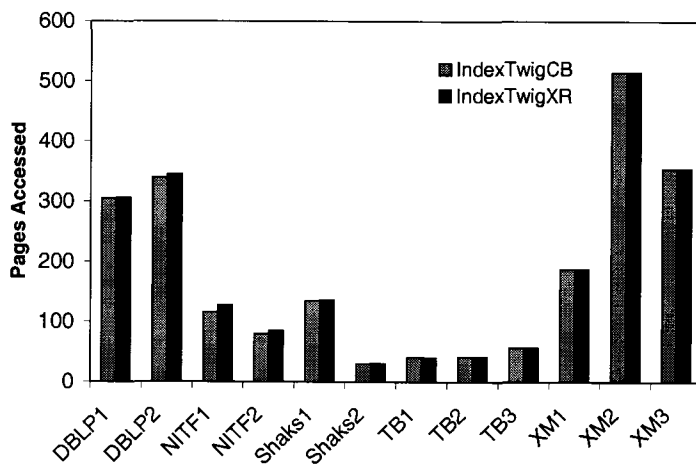


FIGURE 5.10. IndexTwig CB-tree vs. XR-tree

out and output. The answers of other query nodes can be skipped if their existence is of no use for finding target answers.

In IndexTwig, a parent query node passed the result range down to its children. When a query node reached the end of its cursor, IndexTwig used the `FlushAnswer` function with the result range to flush the results of its descendants. After the flush, it is safe to terminate the query processing. The algorithm did not access the rest data that still remained in cursors. In the TSGeneric+ algorithm, the algorithm could end only if all the leaf query nodes reached the end of data cursors. Since there were no result ranges passed down to the descendants for filtering, they had to scan the leaf cursors to the end before finish. This caused unnecessary data access.

IndexTwig passed *min* down to its child query nodes and further to its descendant nodes. In this way, the forward jump (`SJCursor`) could happen among ancestor and descendant query nodes. Unlike TSGeneric+, only the parent-child node pairs were picked for `SJCursor`.

In IndexTwig, the `PickChild` function was used to choose a child to forward and find local answers (extensions). Inside this function, we could make the decision based on heuristics. Current implementation of IndexTwig used the smallest child as the choice hoping the largest jump could happen. Further optimizations could be applied here. For example, data statistics may be used as heuristics to make the choice.

5.6 Summary

In this chapter, we proposed the Containment B⁺-tree(CB-tree) index (an extension of the B⁺-tree) that supports both the *containment query* and the *reverse containment query*. The CB-tree is effective for XML documents without or with a small number of recursions, When there is no recursively nested element in XML data, the CB-tree is the same as the B⁺-tree. We also proposed the *IndexTwig* algorithm for processing XPath queries. The algorithm is flexible in that it works with any index

supporting containment and reverse containment queries. A simplified output model was proposed and used in IndexTwig. Based on this, we further proposed the *Fast Existence Test* (FET) optimization to skip unnecessary data and avoid generating unwanted results. The experimental results showed that the IndexTwig algorithm was highly efficient to process twig queries. It outperformed the TSGeneric+ algorithm by a large factor in many cases.

CHAPTER 6

XML PATH PREDICATE PROCESSING

As introduced in the previous chapter, the existential type predicates in a path expression make the whole query to be a twig pattern, and the IndexTwig algorithm deals with structural joins among elements without considering other types of predicates. In this chapter, we will introduce the techniques to process the Boolean type predicates, especially the inequality predicates.

6.1 Introduction

In XML path expressions, predicates are used to test and filter sequences of element nodes. The definition of predicates in XQuery is shown in the following:

$$\langle \text{Predicates} \rangle ::= ([\langle \text{Expr} \rangle])^*$$

Predicates are enclosed in square brackets. One or more predicates can be applied on an expression, in the form $E_0[E_1][E_2]\dots[E_n]$, where E_0 to E_n are path expressions, and E_1 to E_n are used to filter the sequence returned by E_0 . For each item in a sequence to be filtered for example e , the predicate expression is evaluated using e as the *context item*. The context item is the item currently being processed, for example, if e is a node, e is the starting point for the predicate path expression.

Since a predicate expression such as E_1 does not have to return a boolean value, we need definitions to determine which value of predicates retains or discards the items in a sequence. If the value of the predicate expression is a positive number, the predicate is true if the number is equal to the context position of the context item in the sequence. Note that the first position in a sequence is one. For other values, we

need to get the *effective boolean value* of the predicate. The effective boolean value is `false` for the following value:

- An empty sequence
- The boolean value `false`
- A zero-length value of string
- The numeric value that is equal to zero
- The special float or double value `NaN`

In this dissertation, we divide the predicates into three categories, *existential*, *positional*, and *boolean* types:

Existential Predicate If the value of a predicate is a sequence, we call it an existential predicate. If the sequence is not empty, the effective boolean value is `true`.

Positional Predicate If the value of a predicate is a positive number, it is used to match the context position of the context item. This kind of predicates are called positional predicates.

Boolean Predicate This category includes the predicates generating a boolean value (`true` or `false`) or other atomic value such as string, float or double.

Let us use the same book list example in Figure 1.1 to illustrate the different kinds of predicates. The path expression, `//book[chapter]`, finds all `book` elements with at least one `chapter` element. The predicate in this query is an existential predicate. It uses the existential test to select the `book` elements. If we want the second book in the book list, we can use the expression, `//book[2]`, which uses the positional predicate, `[2]`, to get the second item from the book sequence. For boolean type predicates,

we use the expression, `//book[price < 100]`, as an example. This query picks all books under a hundred dollars.

In Chapter 5, we discussed the path query processing with existential predicates. We captured the character of existential predicates and skipped data by using the simple output model. As for positional predicates, similar techniques can be used to skip the nodes whose positions are larger than the value of a positional predicate. In this chapter, we limit our discussion on the processing of path expressions with boolean predicates that contain element nodes, e.g., `//book[price < 100]`. We propose indexes and algorithms to process the predicates in the form $E_1[E_2 \langle op \rangle \langle value \rangle]$, where E_1 and E_2 are elements, and $\langle op \rangle$ is the comparison operator. This form is the basic form for boolean predicates and other complex predicates can be built using this basic form.

6.2 Related Work

Keyword search can be considered as a form of predicate that can be combined in querying XML data. There have been efforts to integrate keyword search into XML query processing [31, 65]. Florescu and Kossmann [31] propose to extend the XML-QL query language [20] with keyword based search capabilities. To use indexes to facilitate keyword searching, the structure of inverted files is also extended to support full-text indexing with additional information such as the granularity of XML elements, the type of keywords, and the depth of the related element instances. Wolff *et al.* [65] make use of structural information within XML documents in the retrieval process based on a probabilistic model. They propose two index structures, a *structure index* that preserves the hierarchical structure of the underlying data, and a *text index* that supports the evaluation of textual queries. Jiang *et al.* proposed the extensions of the holistic join algorithm to process *OR-Predicates* without decomposing the predicates [39]. Our focus in this chapter is on processing inequality predicates

containing both elements and values.

New index structures and search algorithms have been proposed for performing efficient filtering of XML documents in the selective information dissemination environments [5]. In such systems, the roles of queries and data are reversed. To effectively target the right information to the right users, user profiles are posed as standing queries that are applied to all incoming XML documents in order to determine which users the document will be sent to. The standing queries are written in XPath language [16], which allows predicates in queries. Hence, we briefly summarize the path processing techniques on streaming XML data in the following.

XML filtering systems use event-based parsing and Finite State Machines (FSMs) can provide the basis for highly scalable structure-oriented XML filtering systems. The XFilter system [5] was an FSM-based XML filtering approach, which used a separate FSM per path query to allow all of the FSMs to be executed simultaneously. Based on the XFilter work, Diao *et al.* proposed the YFilter that combines all of the path queries into a single Nondeterministic Finite Automaton (NFA) [23]. YFilter exploits commonality among queries by merging common prefixes of the query paths such that they are processed at most once. Gupta *et al.* proposed to construct a single deterministic pushdown automaton, *the XPush Machine*, from the given XPath filters with probably many predicates [36]. They also described optimization techniques to make the automata more efficient.

For streaming data processing, we scan the data once and process a set of queries against the incoming data. The techniques we propose in this chapter work on XML data repositories. Indexes are build on such data repositories to speed-up query processing. In the next section, we first investigate the use of B⁺-trees for predicates. We show the benefits and disadvantages for different indexing schemes. Then we propose a new index structure and its associated access algorithm in Section 6.4.

6.3 Predicate Processing Using B⁺-trees

In this section, we present two possible methods to build indexes helping the predicate processing. One method is to build the B⁺-tree on element values, such as the values of the `price` elements. The other method is to build the B⁺-tree on the elements in the predicate, such as the `price` elements. We will analysis each method and find out its advantages and disadvantages. Since we evaluate path expressions in the form $E_1[E_2 \langle \text{op} \rangle \langle \text{value} \rangle]$, we need a structural join algorithm to join E_1 and E_2 . We will use the IndexTwig algorithm introduced in Chapter 5 to do this. During the structural join, the E_2 elements are evaluated on the fly according to the predicate condition, and provide the IndexTwig algorithm with a sorted element list to join with E_1 . Note that the IndexTwig algorithm and other sort-merge based structural join algorithms require the inputs to be in sorted order.

6.3.1 Indexing Element

A natural way to deal with the predicates is to build similar indexes as the CB-tree in Chapter 5. Let us use the path expression, Q3,

```
Q3: //book[price < 100]
```

as an example. For this query, we can build indexes on the `book` element and the `price` element. Since the `book` and the `price` elements are not recursively defined, the B⁺-tree will be enough. The index key of the B⁺-tree is the element ID. The values of `price` can be stored in the leaf records of the `price` index. Thus, in each leaf entry, there is a element ID, which is the index key, and a value of the element, for example, the price value of a `price` element. During the query processing of Q3, only the `price` element with a value less than 100 will be provided to the structural join algorithm. We refer to this indexing method as the Element-B⁺-tree method or the E-BT in short.

The elements in the B⁺-tree are stored in increasing order of the element ID. Apparently, a structural join algorithm can directly use the index without further sorting on element ID. For a query such as Q3, there are two factors that affect the performance of the query processing, structural selectivity and value selectivity. The structural selectivity is the join selectivity between the two elements, `book` and `price` for Q3. The value selectivity is the selectivity of the condition in the predicate, which is `price < 100` for Q3. For the E-BT indexing method, when the structural selectivity is high, the IndexTwig algorithm can skip a large amount of elements. Consequently, the checking of the value condition to filter these skipped elements are not necessary. In this case, the E-BT indexing method is expected to perform well. On the other hand, the performance of this indexing method does not change much for different value selectivities. The reason is that the access of an element index record does not depend on whether the associated value satisfies the predicate condition or not. If one element record satisfies the structural condition and the associated value satisfies the predicate condition, the element record should be accessed. Even if the value does not satisfy the predicate condition, we still have to get the record and verify this, since it is an answer of the structural join. When we have a high value selectivity and a low structural selectivity, this method cannot take much advantage of the high value selectivity to skip data. For example, in Q3 let us suppose we have only a few cheap books under 100 dollars, we still have to access all the `price` elements, since every `price` element can be joined with a `book` element in the sample data of Figure 1.1.

6.3.2 Indexing Value

An alternative method is to build the B⁺-tree index on the values mentioned in the predicate. For Q3, a B⁺-tree can be built on the values of the `price` elements. Therefore, the key of the index is the value in the predicate. Since each value may

correspond to a set of `price` elements (books may have the same price), we need a list of element records for each value key in the index. Thus each key in the B⁺-tree corresponds to a list of `price` elements, whose values are the same as the key. So, each leaf entry in the index has a key and a pointer to a list of element records. We refer this indexing method as the Value-B⁺-tree method, or the V-BT for short.

Given a price value, we can obtain a list of element ID's using this B⁺-tree. For equality predicates such as `[price = 10]`, this index can be directly used to support the structural query processing. The algorithms introduced in previous chapters (e.g., sort-merge based algorithms), require accessing the elements in increasing order. To satisfy this requirement, the element ID list of each value key in the B⁺-tree can be stored in sorted order. Thus, this B⁺-tree index on values can support the processing of equality predicates without having to sort the list again.

For inequality predicates such as `[price < 100]`, since there are a range of value keys satisfy the condition, multiple element ID lists will be accessed. In this case, we have to sort these lists into a single sorted list to guarantee the increasing order access of all qualified elements. When the value selectivity is high, only a small amount of `price` element lists are selected from the index, the sorting cost is low. Also because only the qualified `price` elements are accessed and take part in the structural join, a large amount of elements can be skipped even some of them may be joined with the `book` elements if only structural join is used between `book` and `price`. Thus this V-BT method can take the advantage of the high value selectivity to skip data. But when the value selectivity is low, virtually all the `price` elements have to be accessed and sorted during query process. The sorting cost will be prominent. Among the accessed elements, even these elements that can be skipped by the structural join are also unnecessarily accessed. In the next section, we introduce the EVR-tree index that can skip those elements and avoid the high cost of sorting.

6.4 The EVR-tree Index

In the previous section, we presented two indexing methods, the E-BT and the V-BT. We also identified the pros and cons of these methods. In this section, we propose the *EVR-tree* index that takes the advantages of both indexing methods introduced before.

6.4.1 Index Structure

The name EVR-tree stands for the element value R-tree, and it is based on the R-tree index. From the E-BT and the V-BT indexes, we learned that neither of them can make use of both high structural and high value selectivities. The EVR-tree, on the other hand, uses the R-tree to index both elements and values together. There are two dimensions in the EVR-tree indexing space. One is the element ID dimension, and the other is the value dimension. For example, in Q3, one dimension is for the price element ID's, while the other dimension is for the values of the price elements. An illustration of the index dimensions is shown in Figure 6.1. Each dot in the figure represents a record in the EVR-tree. An EVR-tree record consists of a pair of element ID and value as the index key, and other information such as the level of the element in an XML tree. The rectangles in the figure illustrates the leaf level grouping in the EVR-tree. Since the EVR-tree is an application of the R-tree, the insertion and deletion algorithms remain the same as that of the R-tree.

6.4.2 Advantages of the EVR-tree

Ordered Element Accesses As mentioned before, the twig join algorithms require the access of elements in increasing order. In order to be used with the join algorithms, the EVR-tree provides a priority queue to support the ordered access. The idea is similar to the nearest neighbor search algorithm using the R-tree, where a priority queue is used such that the record with the highest priority is accessed first. In the

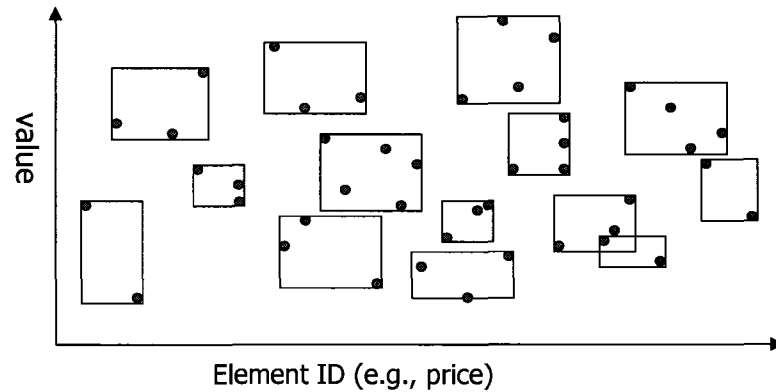


FIGURE 6.1. The EVR-tree Illustration

EVR-tree, since we need to access the elements in increasing order, we can use the element ID to calculate the priority. The smaller element ID the higher priority. In this way, the elements will be accessed through the priority queue in increasing order of the element ID's.

There are two types of items that can be inserted in the priority queue, the element records and the internal index entries. Since we know the element ID of an element record, we can calculate its priority. The internal index entries in the EVR-tree are also needed to be inserted to the priority queue with a different priority calculation. For an internal entry, which points to an index page, the priority is the smallest element ID of the bounding box defined by this entry. Since the bounding box information is stored in an internal entry, there is no extra cost to retrieve the smallest ID. When an internal index entry is popped out from the priority queue, the pointed index page is accessed, and the entries in this page are inserted in the priority queue with their priorities. This index access algorithm is described in Algorithm 11. The priority queue is initialized to contain only the root of the EVR-tree. The input value condition is the condition of the predicate in processing. For example, the value condition for predicate `[price < 100]` will be price value less than 100. An structural join algorithm uses the function to get qualified elements in increasing

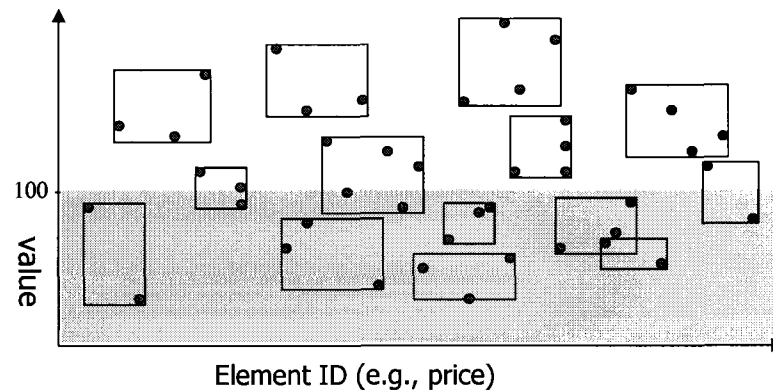


FIGURE 6.2. Skip Values in the EVR-tree

order. In each round of the while loop in the algorithm, an entry is dequeued from the priority queue. If the entry is an internal index entry, the pointed index page is read and the entries in this page are inserted to the priority queue. If the entry is an element record, it is returned to structural join algorithms.

Skip Values The EVR-tree has the advantages of indexing values using the B⁺-tree, which can utilize the high value selectivity to skip unqualified elements and associated disk access. We use Figure 6.2 to illustrate this. The shaded area is the query space in which we need to access the elements for the predicate, $[\text{price} < 100]$. The index pages without any intersection with this query space will be completely ignored without being accessed. The index pages with partial or complete overlaps with the query space may be accessed. This is implemented by Algorithm 11 Line 8 and Line 11. If the bounding box of an index entry overlaps the value query space, then it is inserted (Line 8). Similarly, if an element record satisfies the value condition, it is inserted (Line 11). We will explain next that if the structural selectivity is high the EVR-tree can efficiently support element skips.

Skip Elements As described in the previous chapter, the IndexTwig algorithm will call the function `jumpTo` (refer to Algorithm 9 Line 6) to advance a cursor associated with

Algorithm 11: GetNextElement: for ordered element access of the EVR-tree

Input: *value_condition*
Output: The smallest element in the priority queue
 // the priority queue is initialized to contain the root of the EVR-tree
 // before the first invocation

```

procedure GetNextElement(value_condition)
1 while prio_queue is not empty do
2   entry  $x \leftarrow \text{dequeue}(\text{prio\_queue})$ ;
3   switch the type of entry  $x$  do
4     case internal index entry
5       fetch the index page  $p$  pointed by  $x$ ;
6       if  $p$  is an internal page then
7         for each entry  $e$  in  $p$  do
8           // test using the bounding box of  $e$ 
9           if there are values in  $e$  may satisfy value_condition then
10            enqueue(prio_queue,  $e$ , the smallest ID under  $e$ );
11          end
12        end
13      else
14        //  $p$  is a leaf page
15        for each entry  $e$  in  $p$  do
16          // test the value condition using the value of  $e$ 
17          if the value of  $e$  satisfies value_condition then
18            enqueue(prio_queue,  $e$ , element ID of  $e$ );
19          end
20        end
21      end
22    case element record
23      return  $x$ ;
24  end
25 end
26 return null;
  
```

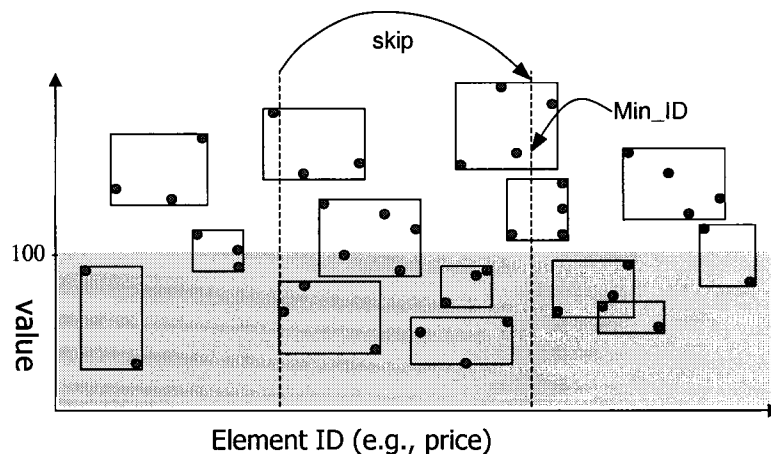


FIGURE 6.3. Skip Elements in the EVR-tree

an index. When the structural selectivity is high, this function can potentially skip many unnecessary index page access. In order to enable the function `GetNextElement` in Algorithm 11 to skip disk pages, we need to pass the parameter min_ID to this function. min_ID is the minimum element ID that is needed by the IndexTwig algorithm. When an entry dequeued from the priority queue is an internal index entry, we can compare the bounding box with min_ID to see if the whole ID range of the internal page is smaller than min_ID . If it is true, we can safely skip this page. If a dequeued entry is an element record and its ID is smaller than min_ID , we discard this element too. In this way, unnecessary disk access and element processing can be avoided. Figure 6.3 illustrates this idea. In the figure, the left vertical dashed line is the ID position of the last returned element, and the right vertical dashed line is the min_ID . By filtering the entries in the priority queue using min_ID , several entries in the priority queue can be skipped as shown in the figure. We present a revised version of the function `GetNextElement` in Algorithm 12, in which Line 5 implements the skipping of internal pages, and Line 15 implements the skipping of elements.

Algorithm 12: The New Version of getNextElement for the EVR-tree

Input: (*value_condition*, *min_ID*)

Output: The smallest element in the priority queue

// the priority queue is initialized to contain the root of the EVR-tree

// before the first invocation

procedure *getNextElement*(*value_condition*, *min_ID*)

```

1 while prio_queue is not empty do
2   entry  $x \leftarrow$  dequeue(prio_queue);
3   switch the type of entry  $x$  do
4     case internal index entry
5       if the ID range of  $x$  is smaller than min_ID then continue;
6       fetch the index page  $p$  pointed by  $x$ ;
7       if  $p$  is an internal page then
8         for each entry  $e$  in  $p$  do
9           // test using the bounding box of  $e$ 
10          if there are values in  $e$  may satisfy value_condition then
11            enqueue(prio_queue,  $e$ , the smallest ID under  $e$ );
12          end
13        end
14      else
15        for each entry  $e$  in  $p$  do
16          // test the value condition using the value of  $e$ 
17          if the value of  $e$  satisfies value_condition then
18            enqueue(prio_queue,  $e$ , element ID of  $e$ );
19          end
20        end
21      end
22    case element record
23      if the ID of  $x$  is smaller than min_ID then
24        continue;
25      else
26        return  $x$ ;
27      end
28    end
29  end
30 return null;

```

6.5 Performance Study

In this section, we present the performance study of the EVR-tree and the two indexing methods using B⁺-trees. We implemented the EVR-tree based on the GiST C++ library [37]. The two B⁺-tree indexing methods used the B⁺-tree implementation of the GiST library. The algorithm we used for structural joins is the IndexTwig join algorithm introduced in Chapter 5.

6.5.1 Experimental Settings

We experimented with two data sets, one synthetic data set and one real-world data set (DBLP). The synthetic data has 1M `price` elements, and various `book` elements from 10K to 1M. Each `book` element contain one `price` element. Hence, different number of `book` elements will give different structural selectivities. The values of `price` elements are generated randomly from 0 to 99. The actual query used for the synthetic data is `//book[price < value]`. The value can be changed from 0 to 99 to choose different value selectivities. For the DBLP data, we used the query with similar format, `//inproceedings[year > value]`. The value of the `year` element can be adjusted from 1990 to 2002 for different selectivities.

We used an Intel workstation with a Pentium 4 1.6GHz CPU running Solaris 8. This workstation had 512M bytes of memory and a 40GB EIDE disk drive (with 7200 RPM and 8ms average seek time). The disk was locally attached to the workstation and was used to store XML data and indexes. We also used the direct I/O feature of Solaris for all our experiments to avoid operating system's cache effects.

6.5.2 Performance Analysis

In our experiments, we measured both the elapsed time and disk I/O count. We noticed that the major cost of the query processing is the I/O cost and the elapsed time is proportional to the I/O cost. So in this section, we will use the I/O count to

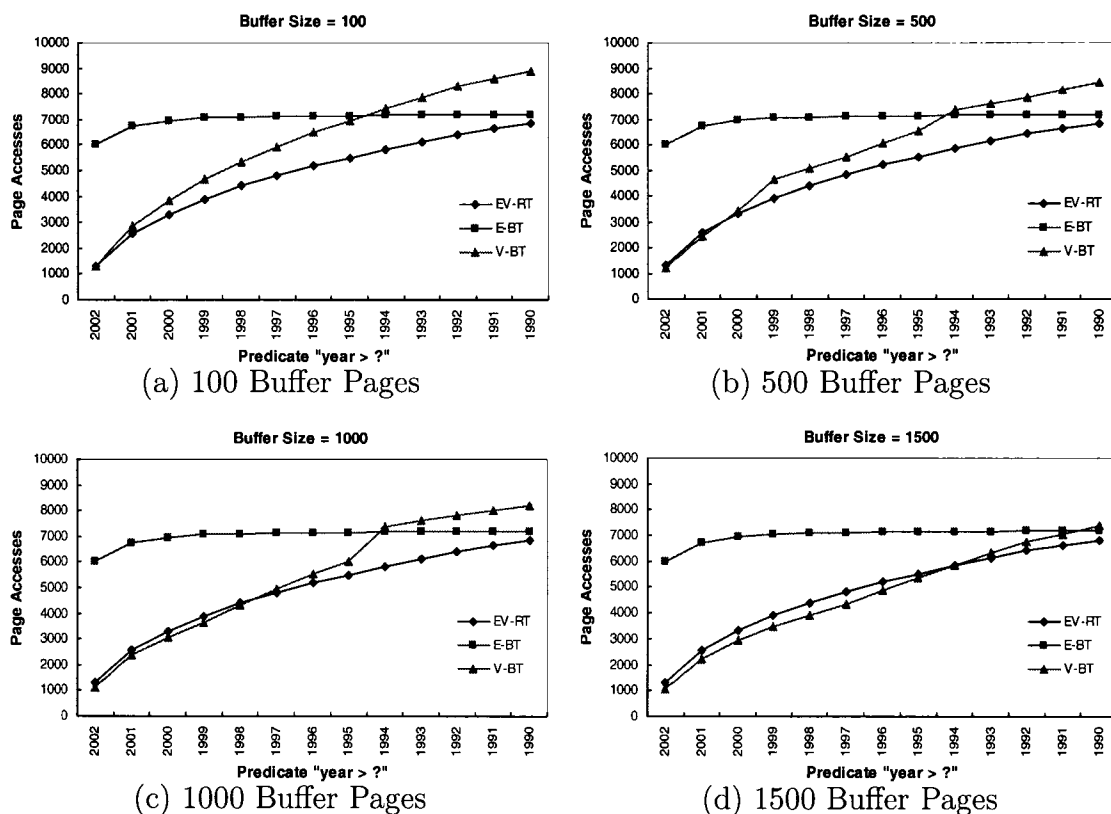


FIGURE 6.4. DBLP Query `"//inproceedings[year>?]"` with Different Buffer Pages and Value Selectivities

show the performance of different indexing methods. For fair comparison, the output cost was not counted.

Figure 6.4 shows the performance of the query, `"//inproceedings[year>?]"`, on DBLP data. Three indexing methods are shown in the figure, the *EV-RT*, the *E-BT* and the *V-BT*, which stands for the EVR-tree, the indexing method on elements, and the indexing method on values. These indexes are built on the `year` elements and/or the values of the `year` elements. As for the `inproceedings` elements, we used the CB-tree. Since there are no recursions among the `inproceedings` elements, the CB-tree is similar to a B⁺-tree. In Figure 6.4, there are four figures for different number of in-memory buffer pages used during query processing. Inside each figure,

we can observe that with the decrease of the value selectivity, more and more elements were selected and used in structural joins. Hence more costs were necessary to process the query. When the value selectivity is high (e.g., `year > 2002`), the V-BT and the EV-RT were significantly better than the E-BT. The reason is that indexing on values can make use of the high value selectivity. Only the qualified elements took part in the structural joins, and others elements were skipped by the indexes. For the E-BT, on the other hand, elements can be skipped only by the structural join algorithm. Since the structural selectivity is almost 100 percent, there were few chances to skip elements.

As mentioned before, the downside of the V-BT is that all qualified elements have to be sorted before taking part in the structural joins. When the value selectivity was low, most of the elements had to be sorted. The V-BT indexing method needs memory buffers to perform the sorting. The sorting cost is prominent if the buffer size is small, as we can see from Figure 6.4 (a). When there were more qualified elements, the performance of the V-BT was worse than the EV-RT. When there were enough buffer pages, as shown in Figure 6.4 (d), the cost of the three methods were close for the low value selectivity. This also demonstrated the low memory requirement of the EV-RT.

The structural selectivity of the DBLP data is fixed and is almost 100 percent. To show the performance of different structural selectivities, we used the synthetic data. Figure 6.5 shows its performance. In this set of experiments, we used 1000 memory buffer pages. The overall trend of performance is similar to that in Figure 6.4. Among the three indexing methods, the EV-RT performed best. We need to note that for the high structural selectivity, e.g., 1% in Figure 6.5 (a), the EV-RT performed especially well. It took the advantages of both high value and high structural selectivities. Also, as anticipated, the performance of the E-BT was better than the V-BT for the high value selectivity portions in all four figures of Figure 6.5.

One interesting phenomenon in Figure 6.5 (a) is that the performance of the E-

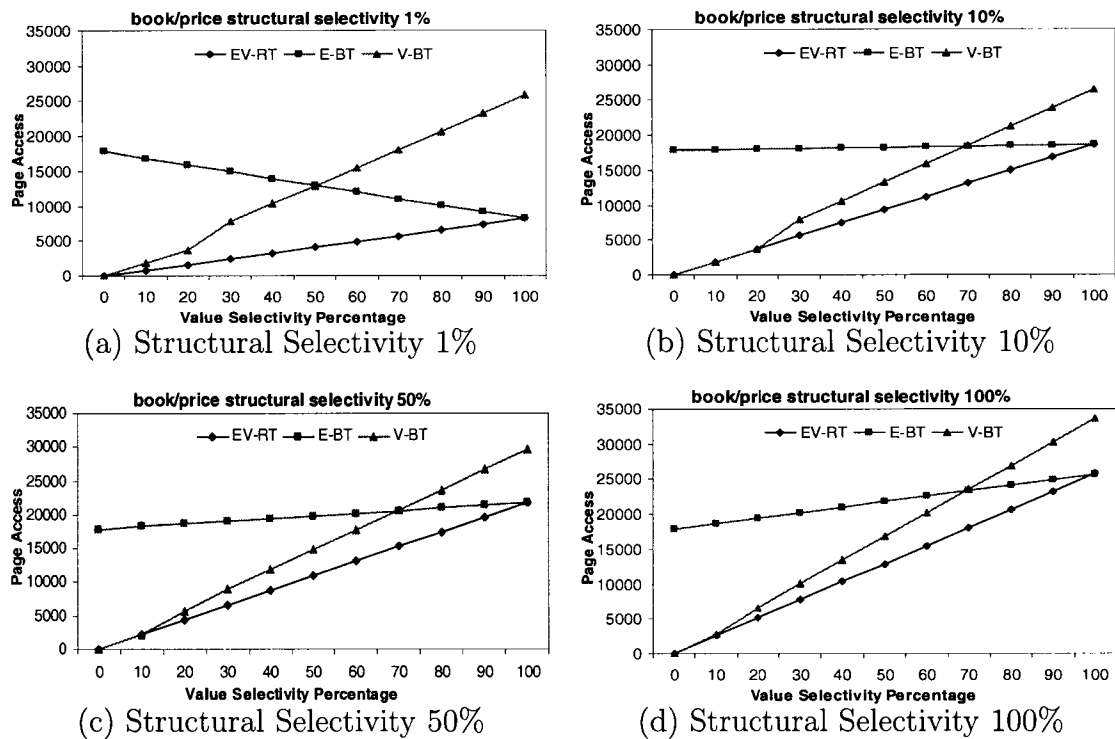


FIGURE 6.5. Synthetic Data Query "`//book[price>?]`" with Different Structural Selectivities and Value Selectivities

BT was getting better with the decrease of the value selectivity. The reason is that when the value selectivity is high, the E-BT index had to sequentially search many leaf pages for the few qualified elements, because there is no index on value. Thus it could not effectively take advantages of the high structural selectivity to skip elements. When there were more qualified elements (value selectivity is low), each such search was faster. Since the structural selectivity is high, the next search could use the structural jump to skip those elements that may be sequentially scanned for the high value selectivity case.

From the above experiments, the EVR-tree demonstrated better performance than indexing individually on values or elements. It took the advantages of high value selectivity and high structural selectivity to skip elements that cannot be joined.

6.6 Summary

In this chapter, we first introduced the predicates of XML path expressions. Then, we proposed the EVR-tree to index the inequality predicates. The EVR-tree combined the advantages of indexing on values or elements individually using B⁺-trees. At the same time, it provided ordered access by using a priority queue. From the experiments, the EV-tree demonstrated better performances over the indexing methods using B⁺-trees. It utilized the high value selectivity and high structural selectivity to avoid unnecessary disk access and unnecessary processing of elements that cannot be matched.

CHAPTER 7

XML PATH PROCESSING USING RDBMS

Relational databases are mature and widely used. Many research activities use existing relational database systems for storing and querying XML data [21, 28, 29, 30, 45, 59, 62, 67, 68]. In this chapter, we describe our implementation of the XML Indexing and Storage System, called *XISS/R*, which is based on the extended preorder numbering scheme and relational databases. For performance comparison, we selected two relational schemas to store XML data. We identify and investigate several important issues that affect the storage and query performance. These issues are introduced in Section 7.2.1. The *XISS/R* system demonstrates approaches for using the extended preorder numbering scheme to store XML data in relational database systems and efficiently evaluating path queries using SQL.

7.1 Introduction

Various approaches to storing and querying XML data have been proposed [8, 21, 30, 62]. Since relational technology is mature and well-developed, using relational databases to store XML data is an important direction to explore. The *XISS/R* system, which to be introduced shortly, demonstrates an efficient approach for using relational database systems to store and evaluate queries on XML data.

As discussed in Chapter 2, the extended preorder numbering scheme [47] is an efficient method for relating semi-structured XML data with structured relational data. Every XML node encoded in this way can be stored in a uniform manner inside a relational database. With the numbering scheme, translating regular path expressions to SQL statements is less complicated and does not involve recursive SQL queries.

In the XISS/R system, we implemented the XML Indexing and Storage System (discussed in Chapter 3) on top of a commercial relational database system. Several key issues involved in storing the XML data using the numbering scheme were identified and relational schemas are generated based upon the choices made in resolving these issues.

The XISS/R system includes the following features.

- A web-based user interface, which enables stored documents to be queried via XPath.
- An XPath Query Engine, which automatically translates XPath queries into efficient SQL statements.
- Multiple relational schemas for comparison.
- Reporting of performance statistics.

Current implementation of the XISS/R supports the combination of the following XPath operators in abbreviated syntax, $a//b$, $a[b]$, a/b , $a[@b = "c"]$ and $a@b$. We need to note that the XISS/R is not yet a full-fledged system. It does not support many features of XPath, such as functions and some axes. Our purpose is to demonstrate the ability to store XML data in relational databases based on our numbering scheme, and to efficiently process core operations of XPath. Designing and implementing the system to support all XPath features would be an important future work to make the XISS/R system usable for real applications.

In previous chapters, we proposed several algorithms and index structures to process path expressions. In the relational database we used, these proposed indexes are not available. It is also an important future work to investigate how to implement the techniques proposed in this dissertation inside relational databases with minimized modifications to them. In our performance study, we only utilized the functionalities

provided by the underlying database. For example, we build B^+ -tree indexes and translate path expressions into SQL statements to do query processing.

7.2 System Description

The XISS/R system consists of three components:

1. A mapping of XML data to relational schema.
2. An XPath Query Engine.
3. A web-based user interface.

The mapping of XML data to relational schema is accomplished by using the extended preorder numbering scheme. We have generated two relational schemas that make best use of this numbering scheme. The XPath Query Engine allows XPath queries to be issued on the relational implementation of the mapping of XML data. Universal access to the system is provided through a web-based interface. The interface allows users to visually interact with the system by communicating with the query engine and displaying results and performance statistics.

7.2.1 Mapping XML Data to Relational Schemas

Mature relational technology can be a useful mechanism for storing and querying XML data. Mapping of semi-structured XML data to a highly structured relational system can be accomplished by using the extended preorder numbering scheme. This numbering scheme provides a method for encoding tree-formed data into integer pairs irrespective of data content. We have developed two relational schema to make use of the extended preorder numbering scheme in accomplishing this task.

The Extended Preorder Numbering Scheme The extended preorder numbering scheme associates each node in an XML document with a pair of numbers, the *extended preorder* and the *range of descendants* ($\langle order, size \rangle$). In a relational schema, these pairs can be stored in conjunction with other node information and used as part of join conditions during query processing. Using this numbering scheme, it is not necessary to attempt to use SQL statements to traverse tree structures in order to process ancestor-descendant joins. This numbering scheme also enables the XISS/R system to store nodes in a uniform format as tuples inside relational tables without losing XML document structural information.

Relational Schema The numbering scheme provides a unified way to store the structural relationships of XML data. However, there are a number of options for storing other necessary data from XML documents alongside such structure data. We investigated several key issues that can affect the storage and query performance:

- How to store element and attribute nodes.
- How to store tag name values.
- How to store value string information for text and attribute nodes.
- For different schemas, what kind of indexes are needed.

Before approaching those questions, let us first have a look at the information we need to store in the relational database. XISS/R requires five pieces of information for each node stored in the system. They are the document ID, the order (also referred to as the node ID) and size of a node in the numbering scheme, the depth of a node in a document tree, the tag-name and the possible text value of a node. In an effort to improve efficiency in processing queries and during export of data, we also stored the parent node ID, the sibling node ID, the first child ID and the first attribute ID for each node.

Element Table	Attribute Table	Text Table	Document Table
Document_ID	Document_ID	Document_ID	Document_ID
Order	Order	Order	Name
Size	Size	Size	
Tag_Name	Tab_Name	Depth	
Depth	Depth	Parent_ID	
Child_ID	Parent_ID	Next_ID	
Next_ID	Next_ID	Value	
Attr_ID	Value		

FIGURE 7.1. Tables in Schema A (Primary keys in bold)

Utilizing the above information we have created two relational schemas, *Schema A* and *Schema B*, that are best suited to implementing XISS/R. Note that using these two schemas, we do not have to consider the the DTD or the XML schema of XML data, since the mappings are directly done on XML data.

Schema A XISS/R divides nodes into three categories: element, attribute and text. Since the set of information that each type of node requires to be described is different, this separation saves space by storing only necessary fields. Schema A, which is shown in Figure 7.1, separates nodes along these lines and is defined as follows:

1. The *Document Table* consists of the **Name** of a document and a unique numerical **Document_ID**.
2. The *Element Table* stores all element nodes.
3. The *Attribute Table* stores attribute nodes. The **Value** stores the attribute value.
4. The *Text Table* stores text nodes (not text values) within the system. **Value** stores the actual text.

In this schema, a Document Table is a simple way to separate the document name from the element, attribute, and text relations and to save space by only storing the

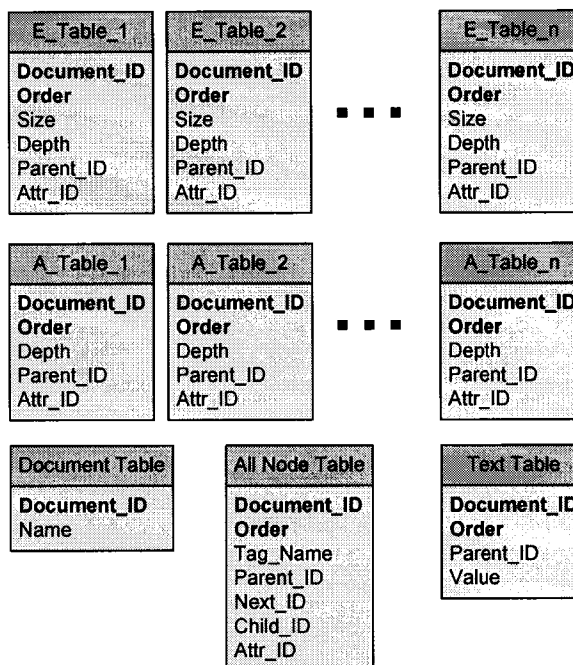


FIGURE 7.2. Tables in Schema B (Primary keys in bold)

expensive document name string information once and inexpensive integer information many times. The element, attribute and text relations store a reference to the numerical ID of the document for each node. In the Element, Attribute, and Text tables, **Order** and **Document_ID** uniquely identify any node within the system. Since all attribute nodes have a corresponding text value (or empty) string, it is stored with the attribute node. This reduces query time.

Schema B Schema B, which is shown in Figure 7.2, goes further than Schema A in separating nodes into different tables. Like Schema A, Schema B separates nodes by their type (element, attribute or text). In addition to this, Schema B horizontally partitions element and attribute by tag-name. An element or attribute table is created for each unique tag-name. All nodes of the same type, from all stored documents, with the same tag name are stored in the same table. For instance, all element nodes with the tag-name ACT are stored in one table. This strategy serves to reduce the

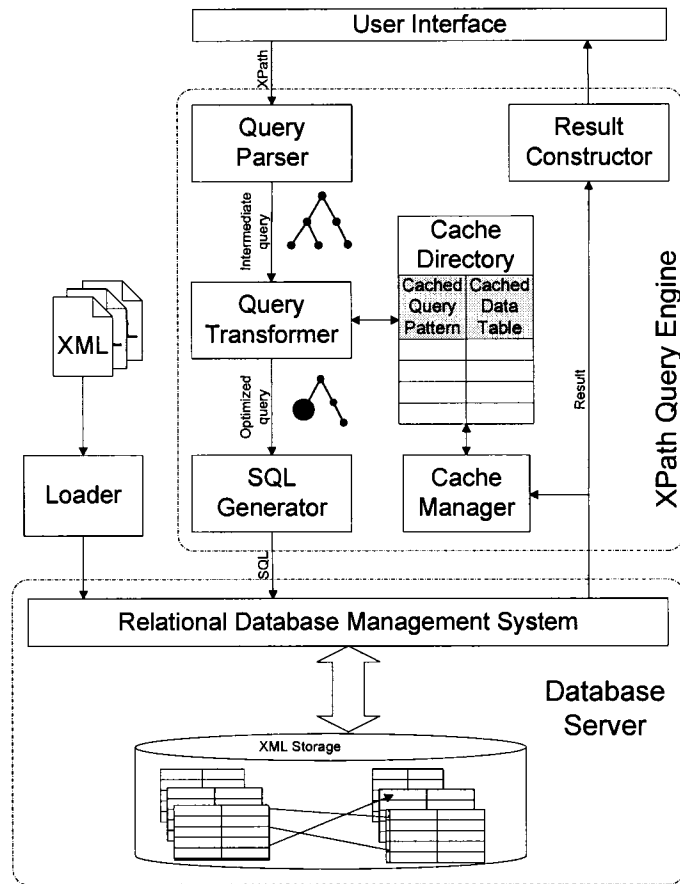


FIGURE 7.3. XISS/R System Architecture

overall query time by making the SQL statements less complicated and by running queries on smaller tables than Schema A. Schema B also contains a table that has necessary information for all nodes to assist in reconstruction of XML documents.

7.2.2 XPath Query Engine

The XPath Query Engine (Figure 7.3), which is located between the user interface and the database server, is the core component of the XISS/R system. The query engine accepts XPath queries and generates SQL statements to send to the database server. Query results from the database server are then formatted and forwarded to the user interface by the query engine.

Upon receiving an XPath query, the *Query Parser* first parses the query and translates it into an intermediate tree-structured format. For example, the query `A[B]/C` will be translated into a tree with node `A` as the root, and `B` and `C` as two children of `A`.

This tree-structured intermediary is then simplified by the *Query Transformer* based on current cache information in the *Cache Directory*. The *Cache Directory* stores result information from previous queries. Each entry in the directory consists of a query pattern in tree-structured intermediate format, and a pointer to its associated result set, which is stored in the RDBMS as a table.

The query transformer matches the current query tree against patterns in the cache directory. If a match is found, the matched part in the query tree will be replaced by a super node. This super node is similar to a normal node except that when parsed into SQL the SQL Generator will evaluate it based on cached result tables. For example, if the result of the query `A/B` is in a cache table, the `A[B]` part in the query `A[B]/C` can be replaced by a super node, which indicates the source data for node `A` should come from the cached table. The query is thus simplified into a two-node query tree. Note that other optimizations can be implemented inside the query transformer.

The simplified query tree is translated into SQL statements by the *SQL Translator* according to the relational schema in use and communicated to the database server. For example, the query `media[@media-type="image"]` will be translated into the following SQL statement when using Schema A:

```
SELECT
    et0.DID as Document, et0.NID as Node_ID
FROM
    elem_tab et0, attr_tab at0
WHERE
    et0.NAME = 'media' and
    at0.NAME = 'media-type' and
    at0.VALUE = 'image' and
    at0.DID = et0.DID and
```

```
at0.PARENT_ID = et0.NID
```

The database server processes the SQL query and returns the result to the XPath Query Engine, whose *Result Constructor* formats the result and sends it to the user interface for display. The result is also sent to the *Cache Manager*, which makes a caching decision based on information such as result set and data size, query pattern or sub-pattern frequency, query processing time, etc.

7.2.3 Web-Based User Interface

The Web-Based Interface allows users to issue XPath queries to the XISS/R system from any location with access to the XISS/R web server. It accepts queries through the HTML form construct, sends these queries to the XPath query engine and receives results in return. Users can choose the amount of results they want displayed at one time and then page through the result set. In addition to the requested portion of the result set, the web query interface also returns usage statistics to users such as the XPath query, the translated SQL query(ies), execution time for each query, and total server side time. After all these are calculated, the statistics are also recorded for future analysis. The recording of these statistics takes a trivial amount of time and does not affect the user.

An example of the query web interface is shown in Figure 7.4. Users can choose the dataset and schema to query. The XPath query and the desired number of results per page are sent to the web server, which interacts with the XPath query engine to process the query. The appropriate portion of the result set, received from the web server is displayed in an additional frame. Here the user can page through the results or peruse performance statistics. Additional queries can be issued at any time.

The screenshot shows the XISS Query Interface in a Microsoft Internet Explorer browser window. The title bar reads "XML Indexing and Storage System: Query Schema A - Microsoft Inter...". The browser's address bar and menu bar are visible. The main content area is titled "Query the Shakespeare data set in Schema A". It features a "Query:" input field with a "Submit Query" button. Below the input field, there is a "Results per page:" dropdown menu set to "5". The "X-Path Query:" is "/PLAY/ACT". The "Sql Query(s) Generated:" is as follows:

```

SELECT
et1.DID as Document, et1.NID as Node_ID
FROM
elem_tab et0, elem_tab et1
WHERE
et0.NAME = 'PLAY' and
et1.NAME = 'ACT' and
et0.DID = et1.DID and
et0.NID < et1.NID and
et0.NID + et0.SIZE_NUM >= et1.NID and
et0.DEPTH + 1 = et1.DEPTH

```

Below the query, it says "Displaying Results 1 through 5:" and shows a table with two columns: "DOCUMENT" and "NODE_ID".

DOCUMENT	NODE_ID
1	67
1	1394
1	3740
1	5374
1	7103

At the bottom, there are navigation links: "<< Prev 1 2 3 4 5 6 7 8 9 10 Next >>". Below that, the "Statistics:" section shows:

```

Total Number of Results = 185
Total Time = 2.8009029626846 seconds.
Time for Query 0 = 0.033360004425049 seconds
Total Query Time = 0.033360004425049 seconds

```

FIGURE 7.4. XISS Query Interface Example

7.2.4 Implementation

Our current implementation of XISS/R uses Oracle 9i as the RDBMS. Documents are parsed and loaded into the database by a program written in Oracle 9i's Pro C/C++ interface. This loader uses the LibXML [66] library to access XML documents such that their structural information can be encoded with the extended preorder numbering scheme.

The web-based interface is implemented with Apache 1.3.24 compiled with PHP 4.1.12. The XPath Query Engine is implemented by PHP scripts that communicate

with Oracle through the Oracle Connection Interface(OCI).

Currently the web interface accepts queries expressed in XPATH 2.0 Abbreviated syntax. The XPath operators, $a//b$, $a[b]$, a/b , $a[@b = "c"]$ and $a@b$ are supported, and can be combined as needed to address stored XML documents. As per the XPath specifications the query result set is composed of pointers that uniquely point to addressed nodes.

We created several schemas to determine whether to store elements in a large node table or store them in separate node tables divided by node name. The overall trend in performance was that schemas using horizontal partitioning were faster. As for whether to store tag names in the node tables or store them in a separate tag name table, we found that since the number of distinct elements and attributes is usually small, the join time between a tag name table and node tables is inconsequential compared to the common total query time.

Unlike a tag name, an XML dataset can and often contains large amounts of distinct value-string information. In worst case scenario, this can be " $2 \times (\text{the number of element nodes}) - 1 + \text{the number of attribute nodes}$ ". With this high percentage of distinct textual values, the amount of space saved is trivial compared to the extra time it takes to access this data. For this reason, the value string information is stored directly in the attribute and text tables.

In addition to the tables in the schemas described before, we also utilized database indexes to accelerate query processing. There are B+-tree indexes on all the primary keys of the tables. In addition, there are B+-tree indexes on name, value and document text information. Also there is a B+-tree index on the `Parent_ID` and `Document_ID` of all text nodes and an index on `Order`, `Size` and `Document_ID` for all nodes.

7.3 Related Work

There have been many research activities that use existing relational database systems for storing and querying XML data [21, 28, 29, 30, 45, 59, 62, 67, 68]. Shanmugasundaram *et al.* [59] have built a prototype system that uses a DTD (Document Type Definition) to convert XML data and queries to relational tuples and SQL queries. Florescu and Kossmann [29, 30] propose a few mapping schemes such as an edge table to store XML data in a relational database. The STORED approach [21] relies on a lossless mapping from a semi-structured data model to a relational model. However, parts of the semi-structured data that do not fit the relational schema are stored in an overflow graph. Thus, the STORED approach is considered a combination of relational and semi-structured techniques.

Fernández *et al.* addressed the problem how to efficiently publish relational data in XML format [28]. Tatarinov *et al.* discussed the XML order issue during storing XML data in RDBMS and translating XML queries. Krishnamurthy *et al.* presented an algorithm to translate path expression queries to SQL in the presence of recursion in XML schemas and queries.

The XRel approach [67] stores paths in relational tables. XPath expressions are translated into SQL queries, which utilize the string matching functionality of a relational database. Zhang *et al.* investigated how to efficiently support containment queries in relational database management systems [68]. In this work, they used the position and depth of a tree node for indexing each occurrence of XML elements

The *XPath accelerator* index structure [34] was proposed to support the processing of XPath axes. Pre- and postorder tree traversal numbering scheme was used. Each element node is represented by a *5-dimensional descriptor*. The XPath axes can be evaluated using these descriptors. Various techniques to minimize the search window and were proposed. Also in this work, relational implementations, including relational schemas and evaluation schemes, were presented.

7.4 Summary

In this chapter, we introduced the XISS/R system, which is an implementation of the XML Indexing and Storage System (XISS) on top of a relational database. The system is based on the XISS extended preorder numbering scheme, which captures the nesting structure of XML data and provides the opportunity for storage and query processing independent of the particular structure of the data. The system includes a web-based user interface, which enables stored documents to be queried via XPath. The user interface utilizes the XPath Query Engine, which automatically translates XPath queries into efficient SQL statements. The query results, query statistics, and the automatically translated SQL statements can be displayed through the web interface.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this dissertation, we first proposed the extended preorder numbering scheme, which determines the ancestor-descendant relationship between nodes in the hierarchy of XML data in constant time. The numbering scheme can adapt gracefully to the dynamics of XML data objects by allocating a numbering region with extra space. Along with the numbering, we introduced the mapping from tree nodes to ranges and the containment property of these ranges.

After encoding XML data using the numbering scheme, XML path queries can be processed using traditional relational database techniques, such as sort-merge based algorithms and partition-based algorithms. Following this, we introduced the XML Indexing and Storage System (XISS) to store and index XML data and to efficiently process regular path expression queries. We identified the major drawback of the conventional methods based on tree traversals—the possible requirement of the extensive search of XML data trees. To avoid this drawback, we proposed to decompose a complex path expression into a collection of basic subexpressions. Each subexpression can be processed either by directly accessing index structures of the XISS system or by applying one of the proposed **\mathcal{EA} -Join** and **\mathcal{EE} -Join** algorithms. For a subexpression having a pair of elements, **\mathcal{EE} -Join** algorithm performs its processing by a two-stage sort-merge operation. The experimental results showed an order of magnitude performance improvement over conventional methods. To process the ancestor-descendant type path expressions, we further proposed the partition-based algorithms, which can be chosen by query optimizer according to the characteristics of the input data. An in-memory range cache was used to hold ranges crossing partition boundaries. The Ancestor Link algorithm can make the best use of memory buffer and take advantage

of the uneven sized inputs.

To process complex path expression with branches, we proposed the Containment B⁺-tree (CB-tree) index and the IndexTwig algorithm. The CB-tree, which is an extension of the B⁺-tree, supports both the *containment query* and the *reverse containment query*. It is an effective indexing scheme for XML documents with or without a small number of recursions. When there is no recursively nested element in XML data, the CB-tree is the same as the B⁺-tree. The proposed *IndexTwig* algorithm works with any index supporting containment and reverse containment queries, such as the CB-tree. We introduced a simplified output model, which outputs only the necessary result of a path expression. Furthermore, the output model enables the *Fast Existence Test* (FET) optimization to skip unnecessary data and avoid generating unwanted results. The experimental results showed that the IndexTwig algorithm is highly efficient to process twig queries. It outperformed the TSGeneric+ algorithm by a large factor in many cases.

We then introduced techniques to process the predicates in XML path expressions. We proposed the EVR-tree to index the inequality predicates. The EVR-tree combined the advantages of indexing on values or elements individually using B⁺-trees. It utilized the high value and/or structural selectivities, and provided ordered element access by using a priority queue. From the experiments, the EV-tree demonstrated better performance over the indexing methods using B⁺-trees.

At the end of the dissertation, we introduced the XISS/R system, which is an implementation of the XML Indexing and Storage System (XISS) on top of a relational database. The system captures the nesting structure of XML data and provides the opportunity for storage and query processing independent of the particular structure of the data. The XISS/R includes a web-based user interface, which enables stored documents to be queried via XPath. The XPath Query Engine automatically translates XPath queries into efficient SQL statements.

In this dissertation, we addressed the ancestor-descendant, twig, and predicate

path query processing. To fully support XPath, other issues such as functions, document order, and wildcard support are important future work. We proposed indexing techniques and query processing algorithms, and evaluated the performances. These information can be used to build a cost model to help optimizers do query optimizations. Building cost models for these proposed algorithms is a necessary future work for query optimizations.

We proposed the XISS/R to store and query XML data. It is an important future work to make the XISS/R system usable to applications. We need to design and implement the system to support all XPath features. In the XISS/R, the indexes and algorithms proposed in this dissertation are not used. It is also an important future work to investigate how to implement the techniques proposed in this dissertation inside relational databases with minimized modifications to them.

Currently, most of XML indexing techniques and query evaluation algorithms are targeting at XPath. How to extend the existing indexing techniques and combine them into XQuery engines is an interesting future research work. Another important part of this research is to do optimization during query evaluations. XQuery is still an evolving standard. Research issues related to optimization are yet to be resolved. There are some information can be used for query optimization. For example, many XML data follows schemas, which can be used in query optimization to simplify queries. XML data statistics are also important sources for query optimization. They can be used in cost models to choose different evaluation plans for XQuery.

As a recommendation of W3C, XQuery is expected to be used widely. More and more Web applications will use XQuery to process XML data. There will be interesting research topics on the integration between Web applications and XML databases. First of all, an XQuery interface for Web applications to access XML databases is necessary. XQuery can be embedded in a host programming language such as Java, or XQuery itself can be a standalone program. A Web application can use this interface to send queries to and get results from XML databases. Since XQuery is a

full-fledged functional programming language, and a query may incur a large amount of computation, an XML database server may encounter more processing load than merely data processing. We may shift processing load from a database server to the client side to increase the throughput of the database server. In order to shift load, trade-offs such as database load status, client computing capability and data transfer cost need to be considered.

REFERENCES

- [1] SIGMOD Record in XML. <http://www.acm.org/sigs/sigmod/record/xml/>, Nov. 2002.
- [2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers Inc., 2000.
- [3] S. Abiteboul, H. Kaplan, and T. Milo. Compact Labeling Schemes for Ancestor Queries. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 547–556, Washington, D.C., United States, 2001. Society for Industrial and Applied Mathematics.
- [4] S. Abiteboul and V. Vianu. Regular Path Queries with Constraints. In *the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 122–133, Tucson, AZ, May 1997.
- [5] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the 26th VLDB Conference*, pages 53–64, Cairo, Egypt, Sept. 2000.
- [6] J. Bentley. Algorithms for Klee’s Rectangle Problems. Unpublished notes, Dept. of Computer Science, CMU, 1977.
- [7] S. Boag, D. Chamberlin, M. Fernández, D. Fernández, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery>, Aug. 2003.
- [8] P. Bohannon, J. Freire, P. Roy, and J. Simon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering*, pages 64–75, San Jose, California, Feb. 2002.
- [9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 Second Edition W3C Recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, Oct. 2000.
- [10] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM-SIGMOD Conference*, pages 310–321, Madison, Wisconsin, June 2002.
- [11] J. Celko. *Joe Celko’s Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann Publishers Inc., 2004.

- [12] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: A Graphical Language for Querying and Restructuring XML Documents. In *Proceedings of the 8th International World Wide Web Conference*, pages 93–109, Toronto, Canada, May 1999.
- [13] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *International Workshop on the Web and Databases (WebDB'2000)*, pages 53–62, Dallas, TX, May 2000.
- [14] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random Sampling for Histogram Construction: How Much Is Enough? In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 436–447, Seattle, Washington, USA, June 1998.
- [15] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th VLDB Conference*, pages 263–274, Hong Kong, China, Aug. 2002.
- [16] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0 W3C Recommendation. Technical Report REC-xpath-19991116, World Wide Web Consortium, Nov. 1999.
- [17] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 271–281, Madison, Wisconsin, 2002. ACM Press.
- [18] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of the 27th VLDB Conference*, pages 341–350, Rome, Italy, 2001.
- [19] R. Cover. The XML Cover Pages. <http://xml.coverpages.org/xml.html>, Feb. 2001.
- [20] A. Deutsch, M. Fernández, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proceedings of the 8th International World Wide Web Conference*, pages 77–91, Toronto, Canada, May 1999.
- [21] A. Deutsch, M. Fernández, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings of the 1999 ACM-SIGMOD Conference*, pages 431–442, Philadelphia, PA, June 1999.
- [22] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proceedings of the 17th VLDB Conference*, pages 443–452, Barcelona, Spain, Sept. 1991.

- [23] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.
- [24] A. Diaz and D. Lovell. XML Generator. <http://www.alphaworks.ibm.com/tech/xmlgenerator>, Sept. 1999.
- [25] P. F. Dietz. Maintaining Order in a Linked List. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 122–127, San Francisco, California, May 1982.
- [26] H. Edelsbrunner. Dynamic Rectangle Intersection Searching. Institute for Information Processing Rept. 47, Technical University of Graz, Graz, Austria, 1980.
- [27] M. Fernández and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proceedings of the 14th International Conference on Data Engineering*, pages 14–23, Orlando, FL, Feb. 1998.
- [28] M. Fernández, W.-C. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. *Computer Networks*, 33(1–6):723–745, June 2000.
- [29] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, Rocquencourt, France, May 1999.
- [30] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
- [31] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *Computer Networks*, 33(1–6):119–135, 2000.
- [32] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd VLDB Conference*, pages 436–445, Athens, Greece, Sept. 1997.
- [33] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus Hash Revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, 1994.
- [34] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transaction of Database Systems*, 29(1):91–131, 2004.
- [35] H. Gunadhi and A. Segev. Query Processing Algorithms for Temporal Intersection Joins. In *Proceedings of the 7th International Conference on Data Engineering*, pages 336–344, Kobe, Japan, Apr. 1991.

- [36] A. K. Gupta and D. Suci. Stream Processing of XPath Queries with Predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, San Diego, California, 2003. ACM Press.
- [37] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proceedings of the 21st VLDB Conference*, pages 562–573, Zurich, Switzerland, Sept. 1995.
- [38] IPTC, the International Press Telecommunications Council. News Industry Text Format. <http://www.nitf.org/>, Sept. 2000.
- [39] H. Jiang, H. Lu, and W. Wang. Efficient Processing of Twig Queries with OR-Predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 59–70, Paris, France, 2004. ACM Press.
- [40] H. Jiang, H. Lu, W. Wang, and B. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engineering*, pages 253–264, Bangalore, India, 2003.
- [41] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, pages 273–284, Berlin, Germany, 2003.
- [42] H. Kaplan, T. Milo, and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 954–963, San Francisco, California, 2002. Society for Industrial and Applied Mathematics.
- [43] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering Indexes for Branching Path Queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 133–144, Madison, Wisconsin, 2002.
- [44] W. E. Kimber. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation, Aug. 1993.
- [45] R. Krishnamurthy, V. T. Chakaravathy, R. Kaushik, and J. F. Naughton. Recursive XML Schemas, Recursive XML Queries, and Relational Storage. In *Proceedings of the 20th International Conference on Data Engineering*, pages 42–53, Boston, MA, Mar. 2004.
- [46] Y. K. Lee, S.-J. Yoo, and K. Yoon. Index Structures for Structured Documents. In *ACM 1st International Conference on Digital Libraries*, pages 91–99, Bethesda, Maryland, Mar. 1996.

- [47] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, Rome, Italy, Sept. 2001.
- [48] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science*, 116:195–226, 1993.
- [49] M. Ley. DBLP Computer Science Bibliography. <http://www.informatik.uni-trier.de/~ley/db/index.html>, Nov. 2001.
- [50] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Rec.*, 26(3):54–66, 1997.
- [51] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the 25th VLDB Conference*, pages 315–326, Edinburgh, Scotland, Sept. 1999.
- [52] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing Semistructured Data. Technical report, Stanford University, Stanford CA, Feb. 1998.
- [53] L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a First Study. In *Proceedings of the 12th International World Wide Web Conference*, pages 500–510, Budapest, Hungary, 2003.
- [54] G. Miklau. XML Data Repository. <http://www.cs.washington.edu/~research/xmldatasets>.
- [55] F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Springer-Verlag, Berlin/Heidelberg, Germany, 1985.
- [56] D. Quass, J. Widom, R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: a Lightweight Object REpository for Semistructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 549, Montreal, Quebec, Canada, 1996. ACM Press.
- [57] P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of the 20th International Conference on Data Engineering*, pages 288–300, Boston, MA, 2004. IEEE Computer Society.
- [58] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the 28th VLDB Conference*, pages 974–985, Hong Kong, China, 2002.

- [59] J. Shanmugasundaram, K. Tufté, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the 25th VLDB Conference*, pages 302–314, Edinburgh, Scotland, Sept. 1999.
- [60] M. D. Soo, R. T. Snodgrass, and C. S. Jensen. Efficient Evaluation of the Valid-Time Natural Join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 282–292, Houston, Texas, USA, February 1994. IEEE Computer Society.
- [61] D. Srivastava, S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering*, pages 141–152, San Jose, California, Feb. 2002.
- [62] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 204–215, Madison, Wisconsin, 2002. ACM Press.
- [63] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 110–121, San Diego, California, 2003. ACM Press.
- [64] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *Proceedings of the 19th International Conference on Data Engineering*, pages 391–402, Bangalore, India, 2003. IEEE Computer Society.
- [65] J. E. Wolff, H. Florke, and A. B. Cremers. Searching and Browsing Collections of Structural Information. In *IEEE Advances in Digital Libraries (ADL'2000)*, pages 141–150, Bethesda, MD, May 1997.
- [66] XMLsoft. The XML C library for Gnome. <http://xmlsoft.org/>, Jan. 2001.
- [67] 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 Transactions on Internet Technology (TOIT)*, 1(1):110–141, 2001.
- [68] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management*

of Data, pages 425–436, Santa Barbara, California, United States, 2001. ACM Press.