Evaluation and Selectivity Estimation of XML Queries

Li Hanyu

Bachelor of Engineering

Zhejiang University, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgement

I would like to express my gratitude to all who have made it possible for me to complete this thesis. The supervisor of this work is Dr Lee Mong Li; I am grateful for her invaluable support. I would also like to thank Associate Professor Wynne Hsu, Professor Ooi Beng Chin and Dr Huang Zhiyong for their advice.

I wish to thank my co-workers in the Database Lab who deserve my warmest thanks for our many discussions and their friendship. They are Ng Wee Siong, Cui Bin, Tang Zhenqiang, Cao Xia, Zhang Zhenjie, Guo Shuqiao, Cong Gao, Zhou Xuan, Wang Wenqiang, Zhang Rui, Dai Bintian, Yang Rui, Shu Yanfeng, Yao Zhen, Lin Dan and Wu Xinyu.

I am very grateful for the love and support of my parents and my parents-in-law.

I would like to give my special thanks to my wife Sun Yu, whose patient love has enabled me to complete this work.

CONTENTS

Acknowledgement **ii**

Summary **x**

1 Introduction **1**

1.1 XML Query Processing 3

1.2 XML Query Selectivity Estimation 7

1.3 Motivation 8

1.4 Contribution 11

1.5 Organization of Thesis 12

2 Related Work **14**

2.1 XML, DTD and Query Languages 14

2.2 XML Query Processing 18

2.2.1 Relational-based Approaches 18

2.2.2 Path Indexes 21

2.2.3 Structural Join Solutions 23

2.3 XML Query Selectivity Estimation 32

3 A Path-Based Approach for Efficient Structural Join and Negation 36

3.1 Introduction 36

3.2 Path-Based Labeling Scheme 37

3.2.1 Path ID 38

3.2.2 Containment of Path IDs 43

3.3 Query Evaluation of Structural Join 46

3.3.1 $PJoin$ 46

3.3.2 $NJoin$ 49

3.3.3 Discussion 50

3.4 Query Evaluation of Negation 53

3.4.1 XQuery Tree 53

3.4.2 $PJoin^+$ 56

3.4.3 $NJoin^+$ 59

3.5 Experiments - Part 1 60

3.5.1 Query Evaluation Performance 61

3.5.2 Update Performance 67

3.5.3 Space Utilization 67

3.5.4 Summary 69

3.6 Experiments - Part 2 70

3.6.1 Storage Requirements 72

3.6.2 Structural Join 74

3.6.3 Negation 88

3.7 Conclusion 92

4 A Statistical Query Selectivity Estimator for XML Data 93

4.1 Introduction . . . 93

4.2 Preliminary . . . 94

4.2.1 Problem Definition . . . 94

4.2.2 Taxonomy . . . 95

4.3 Estimation Method . . . 96

4.3.1 Query Decomposition . . . 96

4.3.2 Summary Statistics . . . 97

4.3.3 Statistics Aggregation Methods . . . 100

4.3.4 Estimation Algorithm . . . 105

4.4 Histogram-Based Estimation . . . 109

4.4.1 Histogram Structure . . . 111

4.4.2 Estimating XML Queries . . . 115

4.5 Experiments . . . 118

4.5.1 NR-NF Estimation Method without Histogram . . . 118

4.5.2 NR-NF Estimation Method with Histograms . . . 123

4.6 Conclusion . . . 129

5 A Path-Based Selectivity Estimator for XPath Expressions with Order Axes 130

5.1 Introduction . . . 130

5.2 Capturing Path and Order Information . . . 132

5.3 Estimating Selectivity of Queries with No Order Axes . . . 135

5.3.1 Path Join . . . 136

5.3.2 Estimating Simple Queries . . . 137

5.3.3 Estimating Branch Queries . . . 137

5.4 Estimating Selectivity of Queries with Order Axes . . . 140

5.4.1 Preceding-Sibling/Following-Sibling Axis 140
5.4.2 Preceding/Following Axis 145
5.5 Data Structures 145
5.5.1 Path ID Binary Tree 146
5.5.2 P-Histogram 147
5.5.3 O-Histogram 149
5.6 Experiments 152
5.6.1 Memory Space Requirement 153
5.6.2 Summary Construction Time 157
5.6.3 Estimation Accuracy of Queries without Order Axes 158
5.6.4 Estimation Accuracy of Queries with Order Axes 162
5.7 Conclusion 165

6 Conclusion **168**
6.1 Summary of Main Findings 169
6.1.1 XML Query Processing 169
6.1.2 XML Query Selectivity Estimation 170
6.2 Future Work 171

LIST OF FIGURES

1.1 Example of XPath Query 3

2.1 Example of XML Data 15

2.2 Example of XML DTD 15

2.3 Interval-based Labeling Scheme 24

2.4 B^+-Tree 26

2.5 XR-Tree 27

2.6 XB-Tree 28

2.7 XML Instance, XSketch and XML Query 35

3.1 Path-Based Labeling Scheme 39

3.2 Storage Structure 42

3.3 Example of $PJoin$ 48

3.4 Example of Exact Pid Set 51

3.5 Examples of Super Pid Set 52

3.6 XQuery Tree. 54

3.7 Example of $PJoin^+$ 57

3.8 Low Ancestor Selectivity . . . 63

3.9 High Ancestor Selectivity . . . 64

3.10 Descendant Selectivity . . . 65

3.11 Levels of Nestings . . . 66

3.12 Update Cost . . . 68

3.13 Space Consumption . . . 69

3.14 Implementation of BLAS . . . 72

3.15 Effectiveness of Path Join . . . 75

3.16 XB-tree Based Holistic Join vs. Path Based Structural Join . . . 78

3.17 Parent-Child Queries . . . 80

3.18 Queries with Value Predicates . . . 82

3.19 Decomposing a Branch Query into a Set of Suffix Queries . . . 85

3.20 BLAS vs. Path-Based Solution . . . 87

3.21 Effectiveness of Path Join+ . . . 90

3.22 TwigStackList¬ vs. Path-Based Negation Join . . . 91

4.1 Classification of XML Queries . . . 96

4.2 Decomposing a General Query into a Set of Basic Queries . . . 97

4.3 An XML Instance . . . 98

4.4 NR and NF Values for Parent-Child Paths . . . 99

4.5 Estimating Frequency of Node N in Query Q . . . 109

4.6 Example of a Skewed XML Instance and its NR-NF Values . . . 110

4.7 Histograms of Paths . . . 112

4.8 Compatible Bucket Sets . . . 116

4.9 Comparative Experiments . . . 122

4.10 Memory Usage with Histograms . . . 124

4.11 Error Rates with Histograms . . . 125

4.12 Histogram-Based Approach vs. *XSketch* 127

5.1 Path Encoding Scheme 133

5.2 Path and Order Information 134

5.3 Example of Path Id Join 136

5.4 Estimating Selectivity of Branch Query 138

5.5 XPath Query with Order Axes 143

5.6 Path Id Binary Tree 146

5.7 P-Histogram 148

5.8 O-Histogram 150

5.9 P-Histogram Memory Usage 155

5.10 O-Histogram Memory Usage 156

5.11 Estimation Error of Queries without Order Axes 159

5.12 P-Histogram vs. XSketch 160

5.13 P-Histogram vs. NR-NF Histogram 163

5.14 Estimation Error of Queries with Order Axes (Branch Part) 164

5.15 Estimation Error of Queries with Order Axes (Trunk Part) 166

Summary

With the fast-growing use of XML data on the Web, optimizing XML queries has become one of the most active and exciting research areas. Developments in query processing and selectivity estimations of XML data are among the major issues since they determine data access methods and the best possible execution plans for complex XML queries respectively. In this thesis, we examine the problem of query evaluation and selectivity estimations of XML queries, and we develop efficient approaches for them.

First, we examine how path information in XML data can be utilized to speed up structural join, which is the core operation in XML query processing. The proposed solution comprises of a path-based node labeling scheme and a path join algorithm. The former associates each node in an XML document with its path type while the latter greatly reduces the cost of subsequent element node join by filtering out elements with irrelevant path types. In addition, this approach is also efficient for an important class of XML queries involving structural anti-join. Comparative experiments demonstrate that the proposed approach is efficient and scalable for queries ranging from simple paths to complex branch queries, and queries involving

anti-join relationships.

Next, we investigate selectivity estimations for XML queries. We design a compact statistical method which extracts two highly summarized information, namely, node ratio and node factor, from every distinct parent-child path in the XML data. When evaluating an XML query, statistical information is recursively aggregated to estimate the selectivity of the target node in the query pattern based on the path independence assumption. Compared with existing solutions, this method utilizes statistical data that is compact, and yet proves to be sufficient in estimating the selectivity of XML queries.

To estimate the selectivity of XML queries with order-based axes, such as *preceding* and *following* axes, we utilize the path-based labeling scheme to collect the path information where XML elements occur and the order information between sibling XML nodes. The summarized path information and order information are then applied to estimate the selectivity of XML queries without and with order-based axes respectively. In addition, we design the path histogram (p-histogram) and the order histogram (o-histogram) to summarize the path information and the order information respectively. To reduce the effect of data skewness in the buckets, both histograms use intra-bucket frequency variance to control their construction. An extensive experimental study on various real-world and synthetic datasets shows that the proposed solution results in very low estimation error rates even with very limited memory space for both XML queries without and with order-based axes.

In summary, this thesis proposes techniques of query processing and query selectivity estimation for XML data. Through an extensive performance study, the proposed solutions are shown to be efficient and easy to implement, and should be helpful for subsequent research in XML query optimization.

CHAPTER 1

Introduction

With the increasing popularity of the World Wide Web and the widespread use of new technologies for data generation and collection, we are flooded with huge amounts of fast-growing data and information. The explosive growth comes from business transactional data, medical data and scientific data, etc. Such data are collected and stored in numerous distributed repositories. Searching for useful information in repositories around the world is beyond human ability without powerful tools. As a result, people typically retrieve such data using search engines like Yahoo [5] and Google [6] which provide full-text indexing services. The user provides one or more key words, and the search engine returns the matching documents which contain these words.

The emerging Extensible Markup Language (XML) Web-standard [8] allows more sophisticated querying of documents. XML allows description of the semantic nature of document components, enabling users not only to make full-text queries, but also to utilize document structure to retrieve more specific data. For example,

we can find the professor at the department of computer science who has the most publications among all staff in the university.

The key observation guiding the design of a search engine that supports structural queries is that an XML document can be viewed as a tree (or graph) whose nodes represent document items and edges denote the relationship between these nodes. Searching for useful information can be achieved by evaluating structural queries posed on XML documents. Given that the cost of scanning XML data to obtain correct results is extremely high when we process a large amount of data, especially for Internet-scale XML documents, successful XML query evaluation hinges on XML query optimization systems.

The XML has become the dominant standard for exchanging data over the World Wide Web due to its flexible self-describing feature. XML query languages, such as XPath [10] and XQuery [7], have been proposed for semi-structured XML data. Both of them use path expressions to traverse irregularly structured XML data to find the sub-structures that match the given query patterns.

With the increasing amount of XML data and the number of XML applications, there is a great demand for efficient XML data management systems for managing complex queries over large volumes of local and Internet-based XML data. As in relational optimization systems, the major issues in XML query optimization systems are query processing and query selectivity estimation.

While complicated query processing engines allow users to directly explore the large amounts of data stored in XML databases, optimizing XML queries with sophisticated path expressions depends crucially on the ability to obtain effective compile-time estimations for the selectivity of these expressions over the underlying XML data. As a result, developing efficient query processing engine and effective query selectivity estimator unavoidably become the core task for building success-

ful XML query optimization systems. In the following sections, we provide the background to these two topics.

1.1 XML Query Processing

The problem of efficient XML query evaluation has received a significant amount of attention in the database community. Consider the XPath query "$book[title = XML]//authors$" as shown in Figure 1.1 that retrieves all the authors of book which has the title "XML". This query can be viewed as a tree-structured query which comprises of a value predicate "$title = XML$" and two structural relations "$book/title$" and "$book//authors$", where "/" and "//" denote the parent-child and ancestor-descendant relationships respectively. Answering this XML query requires we find all matching node instances in the given XML database.

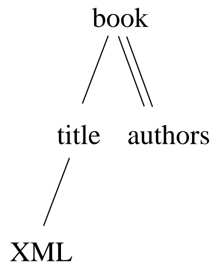


Figure 1.1: Example of XPath Query

An naive solution to evaluate the XML query above is to navigate the entire XML data to find all matching results. Clearly, the cost this method is greatly expensive for huge XML dataset. In the search for effective and efficient query evaluation solutions, different approaches employing different techniques to speed up the query processing have been developed. These techniques can generally be classified into three categories: Relational-based approaches, Path indexes and Structural join solutions. Next, we give a brief overview of these techniques.

Relational-based Approach

Relational database solutions for data storage and query optimization have been well studied for decades. As a result, using a relational approach to store and query XML documents has become a natural "shortcut" for XML query processing since this allows the use of well-established indexing techniques and optimization methods in relational databases. Thus XML query processing is equivalent to evaluating SQL queries in relational databases. In this context, many techniques have been proposed for mapping XML documents into relations and for translating XML queries into SQL queries over those relations.

In [36, 72], the mapping of XML data to a number of relations is considered along with translation of a subset of XML queries to SQL queries. However, the structure of XML data is greatly different from that of relational data. This inherent feature of XML data produces great difficulty when converting XML to relational data. For example, it is very difficult to find an efficient way to store the order information between XML sibling nodes in relations while sibling sequence order is the feature that exists only in tree-structured XML data.

Subsequent work [21, 34, 70] considers the problem of publishing XML documents from relational databases, and [20] studies the issue of updating XML data. However, the fundamental problem of finding a proper way to convert XML data to relations remains. Therefore, designing an index structure on the original XML data is necessary. This leads to the design of path index techniques and structural join solutions.

Path Indexes

XML query languages allow users to navigate arbitrary long paths in a given XML

tree. However, the cost to traverse the XML data entirely is unacceptably high in large datasets. Hence, using structural summaries or path indexes to speed up query evaluation becomes the important issue for XML query processing.

Major path index solutions include DataGuides [38], 1-Index [63], Index Fabric [32] and BLAS [25]. DataGuides [38] and 1-Index [63] summarize raw paths starting from the root to node in an XML document. These index structures do not support branch queries and XML queries involving wildcards and ancestor-descendant relationships efficiently. Index Fabric [32] utilizes index structure Patricia Trie to organize all raw paths, and provides support for "refined paths" which frequently occur in the query workload. These "refined paths" may contain branch queries. However, if a branch query is not included in the "refined paths", then a costly join has to be carried out.

The path index approach BLAS [25] utilizes intervals to represent raw paths and builds a B^+-tree to index these intervals. Given an XML query, BLAS searches matching path intervals to reduce the sizes of candidate element sets. [25] shows that BLAS can perform satisfactorily with suffix queries.

Structural Join Solutions

Structural join is a core operation in many XML query optimization methods. Structural join assumes that the ancestor and descendant nodes involved in a containment XML query, for example "*//article/title*", are provided in two ordered lists. Then a join between these two lists is carried out to find all matching occurrences.

XISS [57] uses a sort-merge or a nested-loop method to process a structural join. This approach scans the same element sets multiple times in case the XML data is recursive. The binary structural join algorithm Stack-Tree [17] resolves the

problem by utilizing an internal stack to store a subset of the data that is likely to be used later. Hence, only one sequential scan is required for each of the lists involved in the join, leading to optimal performance. However, Stack-Tree [17] may still incur unnecessary I/O costs due to the scanning of entire lists, especially in the case that only a small portion of nodes in the lists satisfy the containment relationship. This leads to the design of index-based structural join algorithms.

Major index-based binary structural join solutions include the B^+-tree [28], one dimensional R-tree [28], the XB-tree [19] and the XR-tree [43]. Both the B^+-tree and R-tree approaches are proposed in [28]. They utilize the B^+tree and R-tree respectively to index XML data. As expected, the experiment results show that the B^+-tree approach outperforms the R-tree approach since the structure of R-tree is more suitable for organizing multi-dimensional space data, not one-dimensional interval data proposed for XML nodes. The XR-tree [43] solution further improves query evaluation performance by utilizing stab lists to support more efficiently the operations $findDescendants$ and $findAncestors$ in structural join. The XB-tree proposed in [19] combines the structural features of both the B^+-tree and the R-tree. Compared with the XR-tree, the XB-tree does not store duplicate copies of data. This leads to lower update cost and more efficient space utilization.

These index-based approaches can only deal with binary structural joins which contain just two nodes in the queries. To handle containment queries which involve more than two nodes, holistic twig join methods such as the XB-tree based TwigStack and the XR-tree based TSGeneric are designed in [19] and [44] respectively. They share the same join algorithm but use different underlying data structures. Every element in the query pattern is associated with a stack that stores the possible results. The indexes are useful for skipping sections of the element lists without missing any match. These holistic join solutions treat an XML query as

a whole, thus avoiding the decomposition of a twig query pattern and the merging of intermediate results in most cases.

Finally, note that the classification of these query processing solutions is not rigid. For example, some path index approaches and structural join solutions can also be implemented in relational databases. We do not classify these methods into relational-based techniques because they apply some index structures or access methods which do not exist in standard relational databases.

1.2 XML Query Selectivity Estimation

With the popular use of XML queries, optimizing XML queries with complex path expressions depends crucially on the ability to obtain effective compile-time estimates for the selectivity of these expressions over the XML data. As with a relational database, knowing the selectivity of sub-queries can help identify efficient query execution plans.

Consider the query shown in Figure 1.1 again. We may choose to evaluate the sub-query "*book/title*" first or "*title/XML*" instead. It is obvious that different query execution plans will produce the different intermediate result sizes and thus affect the entire evaluation performance of the query greatly. In this case, effective sub-query selectivity estimation will provide the substantial supports for query optimization.

The existing research of selectivity estimation [15, 26, 58, 62, 64, 65, 66, 83] focuses on XML queries without order-based axes, such as *preceding* and *following*. The methods proposed in [15, 58, 62] are based on the Markov models. [62] stores the frequencies of all paths with length up to k. These values are subsequently aggregated to estimate the node frequency of longer paths. [15] deletes low frequency

paths given space constraints. The loss of information is compensated by employing various algorithms such as Suffix-*, Global-* and No-*. XPathLearner [58] utilizes query feedback to collect statistical information. All these Markov-based solutions are limited to simple path queries since they do not provide for sibling information to be collected.

[15] also proposes a path tree which is structurally similar to *DataGuides* [38]. Low frequency nodes are pruned in the path tree. *XSketch* [64] extends XML tree models in [15] to graphs, and considers both simple paths and branch queries. Based on [64], [66] extends *XSketch* to support queries with value predicates. The most recent work [65] considers building histograms on XML tree models.

[26] develops a method to estimate twig queries. A suffix tree is built for all root-to-leaf paths. Every node in the tree is associated with a hash signature which denotes the set of nodes on the path rooted at this node. The hash signature is used to calculate the frequency of twig queries which are merged from multiple simple paths.

[83] presents a position histogram approach. A two-dimensional *position histogram* is built on either the element tag or element content of each element. A *position histogram join* is then carried out to estimate the query result size based on the node interval containment relationship. Since only containment information between nodes is captured, this approach cannot distinguish between parent-child and ancestor-descendant relationships.

1.3 Motivation

In this thesis, we propose XML query processing and selectivity estimation solutions which offer better performance.

XML Query Processing

The relational database is originally designed for structural relational data, not for semi-structured XML data. Using relational database approaches to store and query XML data requires proper mapping methods to convert XML data and XML queries to relational data and SQL queries respectively. The problem is solved by path index solutions and structural join based approaches. However, structural join based solutions treat each element in the lists involved in the join as an independent unit, and lose the structural relationship between XML elements. This loss of connection between nodes results in the deterioration of query evaluation performance when query selectivity is low.

In XML data, the connection between elements is actually represented by paths which consist of a set of elements. For example, the path query "$//university/department/professor$" comprises the elements "$university$", "$department$" and "$professor$". Therefore, if this connection between elements is considered when evaluating XML queries, the performance of structural join can improve greatly.

However, building simple indexes on raw paths in XML data cannot efficiently process branch queries as analyzed before. In this thesis, we design a novel path-based structural join solution that utilizes bit sequences to capture effective path information to speed up the evaluation of XML queries.

Query Selectivity Estimation

The problem of constructing compact statistical information for flat relational data has received a significant amount of attention. Several effective solutions have been proposed, including histogram [68, 67], random sampling [23, 59] and wavelets [76]. However, estimating the selectivity of tree-structured XML data is a more

complicated and difficult problem.

Most existing XML query estimators support a limited class of query patterns. Markov-based models [15, 38, 58] can only estimate linear path queries since they capture only information on path frequencies. Similarly, the techniques proposed in [78] and [80] also focus only on linear queries. In [83], the *position histogram* cannot distinguish between ancestor-descendant and parent-child relationships. In this thesis, we design a statistical solution that captures highly compact summarized information of paths to estimate the selectivity of arbitrary XML query patterns.

In addition, all existing XML selectivity estimators are designed specifically for XML queries without order-based axes. However, it can be observed that XML queries with order axes are the frequently used query patterns in ordered tree-structured XML data. For example, if a book is organized using XML data, the order of chapters in the book is important and a query can ask for the second chapter of the book. Other examples include data with ordered time domain (temporal XML) and DNA sequences stored using XML.

The selectivity estimation of XML queries with order-based axes is a challenging task due to the huge volume of order information that needs to be captured or summarized. A naive approach to estimating ordered XML queries is to organize sibling XML nodes as a set of sequences and utilize the substring estimation techniques developed for relational databases to calculate the selectivity [22, 41, 51]. However, this approach inevitably faces two problems. First, the underlying data structure of XML data is very different from that of relational data, e.g., an element tag occurring in a query sequence can be imposed with selection predicates from the XML tree (the parent, child, ancestor nodes...). Second, string estimation techniques only process continuous substrings such as $\%ab\%$ while XML queries may require discrete sibling node sequences, for example, $\%a\%b\%$.

In this thesis, a framework for estimating the selectivity of XML queries with order-based axes is also described. To the best of our knowledge, this is the first work to address this needy problem.

1.4 Contribution

This thesis examines major issues in XML query optimization systems. We summarize the main contributions as follows:

- To speed up structural join for XML queries, a novel path-based solution is introduced. It comprises a path-based labeling scheme and a path join algorithm. The former associates every node in an XML document with its path information while the latter greatly reduces the cost of subsequent element node join by filtering out elements with irrelevant path types.

  Besides that, the evaluation of another important class of XML queries, negation (the XML queries containing not-predicates), is discussed. The extensive experimental results show that the proposed method is effective and efficient for both structural join and negation.

- A comprehensive solution to estimate the result size of arbitrary XML query patterns is developed. Highly summarized information, namely, $Node\ Ratio$ (NR) and $Node\ Factor$ (NF), from every distinct parent-child basic path is extracted. When evaluating an XML query, statistical information is recursively aggregated to estimate the frequency of the target node in the query. Compared with the existing solutions, our method utilizes statistical data that is compact and yet proves to be sufficient in estimating the selectivity of queries for regularly distributed XML data.

For skewed XML data, we design histogram structures based on the intervals of XML nodes to maintain detailed information. Experimental results indicate that this structure can lead to more accurate estimation of XML queries.

- A framework to estimate XML queries with order-based axes is proposed. We use the path-based labeling scheme proposed earlier to aggregate the path and order information of XML data. Two compact structures, namely, the *p-histogram* and the *o-histogram*, are constructed to summarize the path and order information of XML data respectively. To reduce the effect of data skewness in buckets, intra-bucket frequency variance is used to control the histogram construction.

  In addition, effective methods to estimate the selectivity of XML queries without and with order axes by using the path and order information collected are developed respectively. An extensive experimental study of the proposed approach is carried out on various real-world and synthetic datasets. The results show that the proposed solution results in very low estimation error rates even with very limited memory space for both XML queries with and without order axes.

Overall, our proposed approaches provide an effective and efficient framework for XML query optimization since they greatly improve the performance of XML query processing and provide accurate query selectivity estimation results.

1.5 Organization of Thesis

The rest of the thesis is organized as follows:

- Chapter 2 introduces related work about XML query processing and selectivity estimation.

- In Chapter 3, the path-based labeling scheme is introduced. Based on it, we discuss the query processing of structural join and negation, and compare the proposed approach with state-of-the-art solutions: the XB-tree based holistic structural join TwigStack [19], the iTwigJoin [24], the path index approach BLAS [25] as well as the TwigStackList¬ [86].

- Chapter 4 presents a statistical method for estimating the result sizes of XML queries by extracting highly summarized information from distinct parent-child paths of XML data. Experiment results indicate that this approach requires a very small memory footprint, and yet proves to be sufficient in estimating query selectivity.

- Chapter 5 develops a framework to estimate the selectivity of XML queries with order axes. We describe how the path and order information of XML elements can be captured and utilized to estimate the selectivity of XML queries.

- Chapter 6 concludes the work in this thesis with a summary of our main findings. We also discuss some limitations and indicate directions for future work.

CHAPTER 2

Related Work

In this chapter, we review the current work on XML query processing and selectivity estimation. The rest of the chapter first gives an overview of the XML, DTD and query languages, and then discusses the existing solutions.

2.1 XML, DTD and Query Languages

XML [8] is rapidly emerging as the dominant standard on the Internet since its self-describing structure provides a simple yet flexible means to exchange data for different applications. In this section, we simply introduce the data model for XML, DTD and XML query languages. Further details can be obtained from the corresponding references.

XML Data Model

XML is a versatile markup language. It is able to label the contents of semi-

structured documents. Figure 2.1 shows an example of XML data which contains the information of a movie [3]. A valid XML document can be viewed as a hierarchical data structure. It starts with a root node, and contains the nested (possibly) child nodes. Internal XML nodes could be in the form of elements or attributes, and leaf nodes may be text nodes. For instance, the example contains the root element node "*Movie*", and it has child nodes "*Title*" and "*Year*", and "*Year*" contains the text leaf node "1999", etc.

```
<Movie>
    <Title>Body Shots</Title>
    <Year>1999</Year>
    <Directed_By>
        <Director>Michael Cristofer</Director>
    </Directed_By>
    <Genres>
        <Genre>Drama</Genre>
    </Genres>
</Movie>
```

Figure 2.1: Example of XML Data

```
<!ELEMENT Movie (Title,Year,Directed_By,Genres,Cast)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
<!ELEMENT Directed_By (Director)*>
<!ELEMENT Director (#PCDATA)>
<!ELEMENT Genres (Genre)*>
<!ELEMENT Genre (#PCDATA)>
<!ELEMENT Cast (Actor)*>
<!ELEMENT Actor (FirstName,LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

Figure 2.2: Example of XML DTD

XML DTD

Document Type Definition (DTD) [9] [1] aims to describe the structure of an XML

[1] In some other research work, DTD is sometimes alternatively referred to as Document Type Declaration or Document Type Descriptor

document. It specifies the XML data structure by listing the names of XML elements and all its sub-elements and attributes. The operators *(zero or more), +(one or more), ?(optional, zero or one) can also be utilized to represent the number of occurrences of elements. For example, the DTD shown in Figure 2.2 describes the XML data structure in Figure 2.1. It specifies that the element "*Movie*" must have child nodes "*Title*" and "*Year*", etc., and the element "*Directed_By*" may contain multiple occurrences of the child node "*Director*".

More recently, XML Schema [12] has been also proposed to describe the structure of an XML document. It is an XML-based alternative to DTD and provides more flexible features to define XML elements. Interested readers may refer to [12].

XML Query Languages

Many XML query languages have been proposed to navigate XML data. Among them, XPath [10] and XQuery [7] have emerged as the *de facto* standard query languages. The core of XPath [10] is the location paths which are utilized to navigate XML documents. XPath [10] has the following syntax:

$$PathExpr \ ::= \ /Step_1/Step_2/.../Step_n$$

$$Step \ ::= \ Axis :: \ NodeTest \ \ Predicate*$$

Given an XPath query, the *Axis* in *Step* establishes the set of XML nodes that are reachable via this axis, where *NodeTest* examines the node name and a set of *Predicates* can be imposed on the nodes. For example, two queries issued on the the XML instance in Figure 2.1 are shown as follows, where "*des* ::" and "*child* ::"

denote descendant and child axes respectively:

$$Q1: \quad /des :: Movie/child :: Title$$
$$Q2: \quad /des :: Movie[/child :: Title(text() = \text{“}Body\ Shots\text{”})]/des :: Director$$

Query $Q1$ retrieves the titles of all movies and $Q2$ searches for the director of the movie with the title “*Body Shots*”. Note that $Q1$ and $Q2$ are referred to as simple (or linear) query and twig (or branch) query respectively since $Q2$ requires that element “$Movie$” must satisfy two outgoing paths “$/child :: Title(text() =$ “$Body\ Shots$”)” and “$/des :: Director$”. For simplicity, queries $Q1$ and $Q2$ can be alternatively expressed as:

$$Q1: \quad //Movie/Title$$
$$Q2: \quad //Movie[/Title \; = \; \text{“}Body\ Shots\text{”}]//Director$$

where descendant and child axes are represented as “//” and “/” respectively.

XQuery [7] is also a powerful query language specifically designed for posing queries against XML data sets to realize its full potential. XQuery is an extension of XPath [10]. Queries represented by XQuery are expressions, and these expressions can be combined, creating extremely powerful queries. XQuery expressions have various formats, including path expression, expressions that use operators and functions, element constructors and for-let-where-order-by-return expressions, which are usually referred to as “$FLWOR$” expressions, etc. Compared with XPath, XQuery provides more powerful and flexible methods to support queries over XML data.

Since XQuery expressions could be the path expressions, any XPath expression that is syntactically valid and executes successfully will simply return the same result in XQuery [7]. As a simple example, we can rewrite the query $Q1$ in XQuery

syntax by using "$FLWOR$" expressions as follows; more examples of XML queries can be found in [11]:

$$Q1: \; For \;\; \$p \;\; In \;\; //Movie/Title$$
$$Return \;\; \$p$$

2.2 XML Query Processing

We roughly classify the current XML query processing solutions into three classes: Relational-based approaches, Path indexes and Structural join solutions. We discuss these techniques below.

2.2.1 Relational-based Approaches

The initial impetus of using traditional relational databases to store and query XML data arises from the fact that we can leverage the mature access methods developed for relational databases over decades, such as the indexing structures: the B^+-tree and the R-tree, and the concurrent control mechanisms, etc. The major literature in this field includes [20, 21, 33, 36, 70, 71, 72, 73].

To work effectively, each of the above techniques must be able to accomplish three tasks: 1) create appropriate tables to store XML data; 2) map XML data to the created tables; and 3) convert XML queries to corresponding SQL queries over these tables. Thus, the XML query evaluation becomes equivalent to evaluating SQL queries in relational databases.

However, an unavoidable problem existing in relational-based approaches is finding an "optimal" (if any) way to map XML data into relations. Due to the different requirements of various applications and the intrinsic complexity of semi-structured XML data, it is almost impossible to design a set of relational tables

which can balance storage cost and query evaluation performance very well for all kinds of applications. For example, a storage-cost optimal solution may generate too many tables since different types of elements are stored separately. This leads to query performance deterioration because many costly join operations have to be carried out when evaluating queries. On the other hand, storing redundant XML elements may improve query performance but it wastes storage space and thus incurs extra update costs.

[36] proposes solutions to map edges in XML into relations. The tables records the Object Ids (oids) of parent and child nodes for all edges. Based on this edge-mapping method, three storage solutions are developed in [36], namely, Edge Approach, Binary Approach and Universal Table. Edge Approach uses one table to store all edges, and one tuple represents one edge. Binary Approach groups all edges with the same child names into one table. Universal Table solution generates a single universal table to store all edges. The tuples in this single table are obtained by performing outer joins on all binary tables in Binary Approach. In other words, each tuple in universal table represents a node-to-leaf path in XML data. As we have discussed, the universal solution can provide the best query evaluation performance among the three approaches, but it contains many redundancies.

[33] designs the STORED system for mapping between semi-structured XML data and a relational data model. The STORED system groups similar XML elements into one table according to their element types. In addition, an "overflow" graph is generated to hold those elements which do not match any generated relational schema. Therefore, the STORED system is a combination of relational and semi-structured techniques. The system focuses on data mining techniques to generate a "good" relational schema, which aims to minimize disk space consumption and reduce query evaluation cost if a query workload is available. It finds building

such a cost-based optimization system to be an *NP*-hard problem in the size of XML data, and employs a heuristic algorithm to generate tables.

In [72], three strategies are proposed to map XML data. They are Basic Inlining, Shared Inlining and Hybrid Inlining. These mappings differ from one another in the degree of redundancies. The most redundant one is Basic Inlining, which stores each distinct element tag as a table, and this table contains all descent nodes of the element tag as attributes in the table. Shared Inlining avoids the drawback in the basic technique by representing one XML element node exactly in one table while the Hybrid Inlining solution attempts to find a balance between Basic Inlining and Shard Inlining methods. In [72], the authors also show that these mapping techniques are more efficient than others when evaluating certain XML queries.

[73] proposes a solution similar to inverted-list. The nodes in XML are stored as regions, and paths are represented as strings in relations. This method is also somewhat similar to structural join, which we will discuss later. However, the authors do not explore join methods between elements (attributes). Thus we classify this work as a relational-based solution. In this field , some later studies [20, 21, 70, 71] consider the problem of publishing relational data as XML.

We highlight here that the above classification of query processing methods is not rigid. For example, some structural join or path index methods can also be implemented in a relational database. We do not classify them as relational-based techniques because the focus of these solutions is not the mapping methods. They provide some extra indexing structures or access methods which do not exist in standard relational databases. Next, we discuss path index techniques.

2.2.2 Path Indexes

Many database researchers have developed path indexes to speed up XML query evaluation by restricting the search space to a portion of the XML data. Among them, DataGuides [38] and 1-Index [63] are conceptually similar. Both of them build the summarized graph structure to index XML paths. In these graphes, each node represents a path instance which is the concatenation of all node tags occurring on the root-to-node path in the summarized graph. Therefore, simple path queries can be simply evaluated by searching the summarized graph and then retrieving the object ids associated with the nodes.

However, DataGuides [38] and 1-Index [63] suffers two problems. First, they cannot efficiently process simple queries with partial matching due to the exhaustive search on the entire index structure, such as XML queries starting with descendant axes, e.g., $//A/B/C$, or queries containing “ * ” elements. Second, DataGuides [38] and 1-Index [63] do not provide direct support for branch queries. Thus, costly join operations between intermediate results of simple queries must be performed when evaluating branch queries.

APEX [30] presents a solution to handle the partial matching problem above. It consists of two structures: a summarized graph and a hash tree. The hash tree includes all the nodes in the graph, which are called *hnodes*. Each *hnode* contains a hash table and the entries in the hash table point to other *hnodes* or the nodes in the summarized graph. When evaluating the partial matching simple XML queries, APEX first searches the hash table (possibly multiple times), then according to the results, directly locates the nodes in the summarized graph. This procedure thus avoids searching the entire graph. In addition, APEX extracts frequently used paths from the XML data to guide the construction of the summarized graph.

Index Fabric [32] utilizes the index structure Patricia Trie to organize all the

root-to-node raw paths in the XML data. The raw paths are encoded by using strings, and these strings are inserted into Patricia Trie. Besides raw path, Index Fabric also supports the "refined paths", which are the queries frequently occurring in the query workload. These "refined paths" can contain branch queries. However, if a branch query is not included in the "refined paths", then a costly join has to be carried out.

[47] studies the problem of building indexes to cover branch XML queries. [47] shows that Forward and Backward-Index (F&B-Index) can cover all branch path expressions. Different from DataGuides and 1-Index which group XML nodes with the same incoming paths, F&B-Index groups nodes with the same incoming and outing paths. Thus, it can effectively handle branch queries as well as simple queries. However, an unavoidable dilemma is the size of the F&B-Index being too big to be useful. To solve this problem, [47] proposes a scheme to explore tradeoff between size of indexes and size of queries these indexes can cover.

The work in BLAS [25] also utilizes path information (p-labeling) to reduce the search space for XML elements. BLAS uses XPath to describe query patterns and employs integer intervals to represent all possible suffix paths (paths that optionally start with descendant axis followed by a set of child axes). Hence, BLAS performs best for suffix queries. However, for branch queries and simple queries involving the ancestor-descendant relationship, BLAS must decompose them into a set of suffix queries and carry out join operations to "stitch" the intermediate results.

Both the BLAS approach and our proposed path-based solution in this thesis perform the operations on the paths to pre-filter out unnecessary elements. The core difference between them is that our method utilizes bit sequences, which contain more information than intervals, to denote paths that actually occur in the XML datasets. As a result, our proposed solution yields optimal performance for

simple queries, which are a superset of suffix queries, and produces better results on branch queries than BLAS does. We explain the details of this comparison in the experiment section of Chapter 3.

2.2.3 Structural Join Solutions

Structural join is now considered a core operation in XML query processing. The existing structural join solutions rely on an efficient numbering scheme to quickly determine the relationship between XML nodes. We shall first provide the background to the major numbering schemes used in XML, and then discuss the previous work on structural join.

Numbering Schemes

[57] and [87] propose the interval-based numbering scheme to label XML nodes. Each node is associated with an interval of the format $(start, end)$. For any two given nodes x and y, x is the ancestor of y if and only if the interval of x contains that of y, that is, $x.start < y.start < y.end < x.end$. The interval labels of XML nodes can be assigned by carrying out a depth-first tree traversal (see Algorithm 1). During the procedure of tree navigation, each node is attached with a number when it is visited and this number is increased each time. Note that each node is accessed twice in Algorithm 1, thus an interval is finally associated with each node. For example, the XML nodes of a file system in Figure 2.3 are labeled by using intervals. However, such a statistic interval-based labeling scheme cannot efficiently support XML data update. Although some space can be reserved when assigning intervals (as shown in Figure 2.3), part of or even an entire XML document needs to be re-labeled when update occurs.

[31, 46] designs the prefix-based labeling scheme to process dynamic XML data.

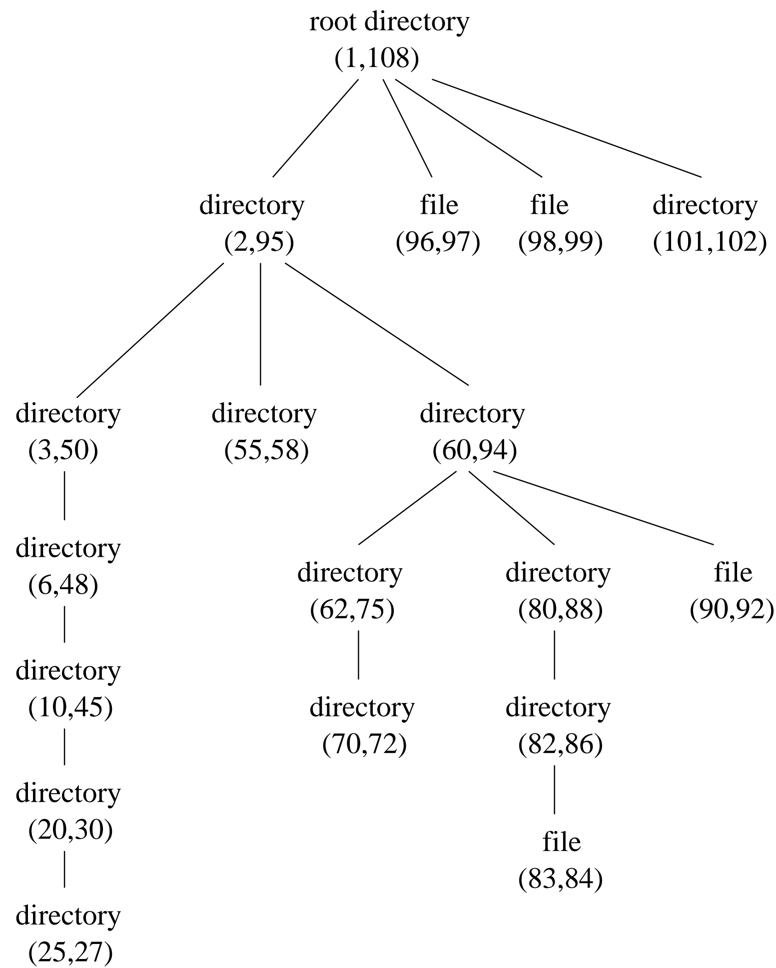


Figure 2.3: Interval-based Labeling Scheme

Algorithm 1 Interval Assignment

$counter = 0;$

$Assign(Node\ node)\{$
$counter++;$
$n.start = counter;$
$child = node.firstChild();$
while $child != NULL$ **do**
  $Assign(child);$
  $child = child.nextSibling();$
end while
$counter++;$
$n.end = counter\ \};$

In this system, the label on an ancestor node is a prefix of the labels of its descendant nodes. In these prefix-based labeling schemes, new nodes can be inserted into the XML structure without affecting the labels of the existing nodes. The later work in [82] utilizes prime numbers to label XML nodes, and it further solves the problem of order update between sibling nodes.

Structural Join

In this part, we use interval-based labels as an example to explain structural join. Structural join solutions can take advantage of the numbering schemes discussed above, that is, quickly determining the relationship between any two given nodes. In such a system, the intervals of all elements (attributes) with the same tags are stored in an ordered list. When evaluating a query such as $A//D$ where A and D denote the ancestor node and descendant node respectively, we need to find all matching pairs of elements (a_i,d_j) in the lists for A and D which satisfy the interval containment relationship. This class of queries are also specified as containment queries [87].

Both [57] and [87] develop join algorithms similar to the standard sort-merge to evaluate containment queries. [57] designs two basic join algorithms according to the different data types: *EE* join (element with element) and *EA* join (element with attribute), while [87] proposes the *MPMGJN* algorithm. However, since XML can hold nested element structures (recursive elements), *EE* join and *MPMGJN* must carry out a nest loop on the parts of the lists for elements involved in the query patterns. This leads to performance deterioration especially for highly recursive XML data due to the multiple scans on the input data. In addition, [57] shows that the nest loop can be safely avoided in *EA* join since recursive definition is only valid for XML elements, but not attributes.

The state-of-the-art structural join solution, Stack-Tree [17], also takes two ordered lists as input to evaluate containment queries. By maintaining an in-memory stack to store the ancestor nodes which may be used later, the two ordered lists are scanned only once. This greatly improves the performance of structural join. However, Stack-Tree may still incur unnecessary I/O cost for highly selective queries since each element in the lists must be accessed before the join can be carried out.

A simple yet effective approach to skip unnecessary data during scanning is to construct indexes on the input lists. These indexes aim to efficiently support functions such as *findDescendants* and *findAncestors* that are needed in structural join. Major index-based solutions for structural join include the B^+-tree [28], the XB-tree [19], and the XR-tree [43].

The B^+-tree [28] solution simply builds the standard B^+-tree on the start points of intervals. Figure 2.4 shows the approach which constructs a B^+-tree on the start points of the *directory* element intervals shown in Figure 2.3. By using B^+-tree range search, this solution can efficiently support the $findDescendants$ operation in structural join.

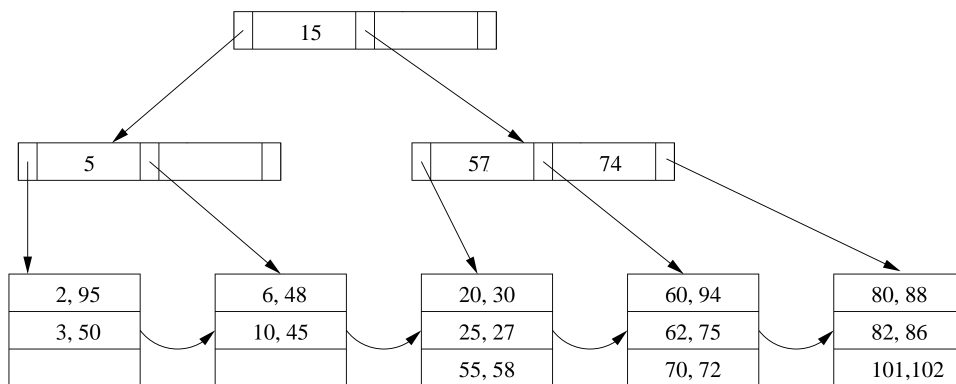


Figure 2.4: B^+-Tree

The XR-tree [43] is essentially a B^+-tree that is built on the start points of

element intervals. Figure 2.5 shows an example of the XR-tree that has been constructed for the *directory* elements in Figure 2.3. Every non-leaf node in the XR-tree is associated with a stab list. The stab list stores the intervals of element entries that can cover any key in the non-leaf node. To facilitate search in the stab lists, each key in the non-leaf node is also associated with the first element interval in the primary stab list that contains the key. Note that besides storing an element e in the leaf node, the XR-tree also stores the element in the stab list of the top-most internal node that contains a key k such that $e.start \leq k \leq e.end$. Compared with the B^+-tree, the XR-tree approach can further efficiently support $findAncestor$ operations by collecting the ancestor intervals stored in the stab lists.

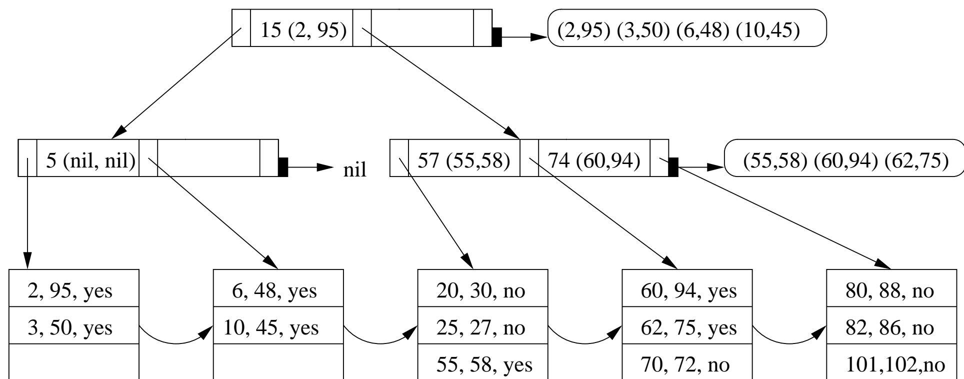


Figure 2.5: XR-Tree

[19] puts forth a proposal of the XB-tree which combines the structural features of both the B^+-tree and the R-tree. The XB-tree first indexes the pre-assigned intervals of elements in a tree structure. From this perspective, the XB-tree is similar to a one-dimensional R-tree. Next, the XB-tree organizes the start points of the intervals in the same way as the B^+-tree does.

Figure 2.6 illustrates the XB-tree that is constructed for the *directory* elements in Figure 2.3. Each internal node maintains a set of regions that completely include all regions in its child nodes. The regions in the nodes of the XB-tree may overlap

partially. However, it differs from the R-tree in that the start points are sorted in a strictly increasing order. In contrast to the XR-tree, the XB-tree does not store duplicate copies of data. This leads to lower update cost and more efficient space utilization.

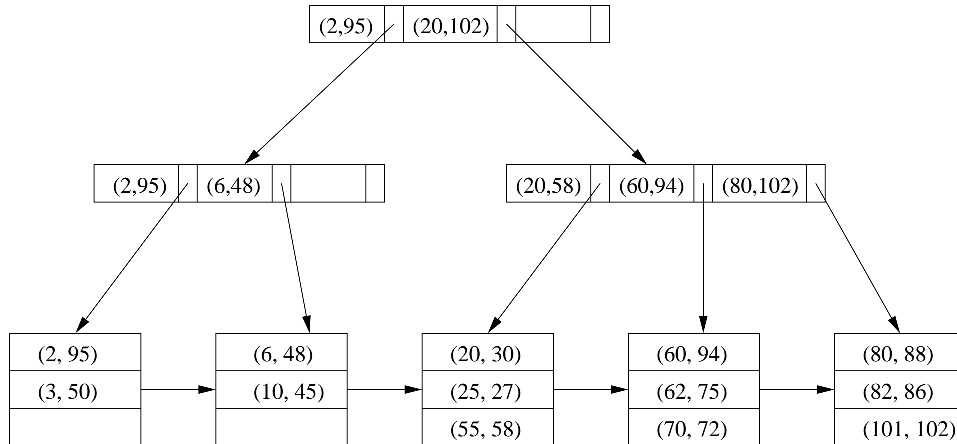


Figure 2.6: XB-Tree

All the index-based solutions above focus on binary structural join, that is, the query pattern only contains two elements. The later holistic solutions extend the binary structural join to holistic twig join [19, 44, 60, 24].

The holistic solutions TwigStack [19] and TSGeneric [44] aim to process XML queries involving more than two nodes. Both approaches avoid decomposing a twig query pattern into a set of binary joins. With a chain of linked stacks to store the results of each root-to-leaf path, they do not produce a large volume of intermediate results compared with binary join based solutions, leading to the I/O optimal performance for most queries among all sequential scan algorithms. In addition, these two approaches propose different indexing structures XB-tree and XR-tree respectively to speed up the query evaluation by utilizing the same join method.

While TwigStack [19] is I/O optimal for the queries with only ancestor-descendant

relationship, it still may produce large intermediate results for the queries with parent-child edges. To overcome this limitation, [60] proposes an upgraded holistic join algorithm, namely TwigStackList. The novel feature in TwigStackList is to look-ahead read some elements in the streams involved in the query pattern and catch limited number of these elements to some lists in the main memory. It shows that the number of elements in the lists is bounded by the length of the longest path in XML data and this method can produce less intermediate result sizes compared with TwigStack for the queries with mixed ancestor-descendant and parent-child relationship.

In TwigStack algorithm [19], each distinct element tag is associated with a stream containing all elements with this tag. This can be viewed as that an XML document is partitioned based on the element tags. In [24], the authors propose a novel holistic join algorithm iTwigJoin which can work correctly on any streaming (partitioning) method of XML data. The iTwigJoin [24] studies two XML streaming schemes: Tag+Level and Prefix (The XML data is partitioned according to Tag+Level and Prefix respectively) and shows that choosing the proper partition methods may greatly affect the query evaluation performance by pruning the irrelevant portions of streams. In Chapter 3 of the thesis, we develop a novel labeling scheme and partition the XML data by using the labels. We demonstrate that our method can be seamlessly combined with TwigStack or iTwigJoin to efficiently evaluate XML queries.

Negation

We note that there is another important class of queries in XML data, negation, which evaluates the negated containment relationship between elements. For example, it is needed when we want to archive publications that have not been cited

before, or when an employer wants to filter out job applicants who do not have any previous working experience. The following query (expressed by using XQuery and XPath respectively) finds all the professors from the computer science department who have *not* received any student complaints. The relationship between $professor$ and *complain* is negated containment (or not-predicate in XPath).

$$XQuery$$

$$For\ \ \$p\ \ In\ \ //dept[/name = \text{“}CS\text{”}]//professor$$

$$Where\ \ count(\$p/complain) = 0$$

$$Return\ \ \$p$$

$$XPath$$

$$//dept[/name = \text{“}CS\text{”}]//professor[Not(/complain)]$$

[16] examines how the binary structural join method can be employed to evaluate negation. This solution is similar to the *Stack-Tree* [17] algorithm. That is, descendant nodes are scanned, and ancestor nodes that do not contain the descendant nodes are popped from the stack and output. The drawback in [16] is that much unnecessary data will be accessed especially when the selectivity is low. In addition, the index techniques proposed for structural join (B^+-tree [28], R-tree [28], XB-tree [19], etc.) cannot be directly applied to process negation since they are designed to capture the containment relationship between elements, but not the negated containment relationship.

Evaluating the complicated XML queries (not binary queries) with negation is also a challenging problem. A naive method is to decompose the given query pattern into a set of sub-queries which do not contain negation and evaluate them individually. After that, the evaluation results of these sub-queries are merged to obtain the final result. Obviously, this approach is not efficient since it may produce

a large volume of intermediate results.

To solve the problem above, [45] proposes an algorithm called PathStack¬ to evaluate holistic path queries. PathStack¬ [45] is a generalization of PathStack [19]. While PathStack [19] associates each query node a stack to temporarily store the possible results, PathStack¬ associates every query node with either a regular stack or a boolean stack according to the position of the node in the query. The novel feature of boolean stack is that the item in stack contains a boolean variable "satisfy" to indicate whether this item satisfies the sub-path rooted at this node or not. By using the boolean stacks, PathStack¬ avoids decomposing holistic path queries with negation and thus improves the negation query evaluation performance.

Similarly, TwigStackList¬ [86] extends the TwigStackList algorithm proposed in [60] to process holistic twig negation queries. Different from TwigStackList [60], TwigStackList¬ [86] associates each projected node (output node) with a stack and a list, and maintains a list only for each non-projected node (non-output node) since non-projected nodes do not contribute to the final results. In addition, TwigStackList¬ introduces the concept "Negation Children Extension" to check weather a query node satisfies the sub-tree rooted at this node. As a result, TwigStackList¬ also avoids decomposing holistic twig negation queries and leads to the optimal I/O costs.

In this thesis, we design a novel path-based approach to efficiently evaluate both structural join and negation.

2.3 XML Query Selectivity Estimation

The optimization of XML queries requires an accurate and compact structure to capture the characteristics of the underlying data. In this section, we will give an overview of the existing work on XML query selectivity estimation.

Markov-based methods [15, 58, 62] are proposed to estimate the selectivity of simple queries, which are usually represented by XPath. In this approach, the frequencies of all distinct paths in the XML data of length up to k are collected in a table, where k is called Markov order. Obviously, this table provides the correct selectivity of queries of length up to k. To estimate the selectivity of query expressions with longer length m, $m > k$, we can utilize the frequencies of shorter paths based on the proper assumption.

Suppose we store the selectivity of all distinct paths up to length 3 ($k = 3$). The frequency of query $a/b/c/d$ of length 4 can be calculated as:

$$f(a/b/c/d) = f(a/b/c) * prob(d|a/b/c)$$
$$prob(d|a/b/c) = f(a/b/c/d)/f(a/b/c)$$

where $f(a/b/c)$ and $prob(d|a/b/c)$ denote the frequency of path $a/b/c$ and the probability of element d occurring in the path $a/b/c$ respectively. Since the value of $prob(d|a/b/c)$ is not provided by the system, we assume that an element in the path depends only on the $m-1$ elements before it. Thus, we can obtain:

$$prob(d|a/b/c) \approx prob(d|b/c)$$
$$prob(d|b/c) = f(b/c/d)/f(b/c)$$

As a result, it is easy to get:

$$f(a/b/c/d) \approx f(a/b/c) * \frac{f(b/c/d)}{f(b/c)}$$

Among the Markov-based solutions [15, 58, 62], [62] directly applies the method introduced above. [58] utilizes query feedback results to collect the statistical information. [15] deletes low frequency paths given memory space constraints. This loss of information is compensated by employing various algorithms such as Suffix-*, Global-* and No-*. Here, the * denotes the deleted low frequency paths. For example, Suffix-* may use path “A/*” to represent the paths “A/B” and “A/B/C” while “B” and “B/C” are low frequency suffix paths. Global-* uses * to denote all deleted paths and No-* does not maintain the information of deleted paths.

[15] proposes an additional solution, path tree, to estimate simple queries. Path tree is structurally similar to *DataGuides* [38]. That is, each node in the path tree represents a path starting from the root to this node, and each node is associated with the frequency of the path it represents. Similarly, low frequency nodes are pruned from the path tree and corresponding compensation algorithms, such as Sibling-*, Level-*, Global-* and No-* are developed.

[26] is the first work to estimate twig queries (bracnch queries). A suffix tree is built for all root-to-leaf paths. Every node in the tree is associated with a hash signature which denotes the set of nodes occurring on the path rooted at this node in XML data. Given a twig query, it is first decomposed into a set of sub-queries which are called twiglets. The number of matches of each twiglet is calculated by using the hash signatures in the suffix tree. After that, the twiglet estimation results are combined to compute the selectivity of the given twig query.

[83] presents a position histogram approach. It utilizes the interval-based number scheme to label XML nodes. A two-dimensional *position histogram* is then built on each element tag, and element interval values are mapped into the predefined

grids over the *position histogram.* Given an XML containment query of the format $A//D$, a join between the *position histograms* of elements A and D is then carried out to estimate the query result sizes based on the interval containment relationship. However, this approach suffers two problem: (a) Since only containment information between nodes is captured, this approach cannot distinguish between parent-child and ancestor-descendant relationships. (b) It is not clear how this solution is to be extended to the XML queries containing more than two elements.

[64] proposes the *XSketch* model for estimating branch queries. *XSketch* also builds the summarized graph structure to represent XML data. The novel feature in *XSketch* is that the edges in the summarized graph are optionally associated with Backward (**B**) or Forward (**F**) marks or both (**BF**). The concept of **B** (**F**) edges are similar to that of the Backward-Index (Forward-Index) we have introduced in the section on query processing. That is, all instances of the parent node of an **F** edge must be the parents of the instances of the child node of this edge in XML data. Note that this is different from the XPath tree which only groups nodes with the same incoming paths. Based on the **BF** edges and proper assumptions, *XSketch* can estimate branch queries as well as simple queries. Figure 2.7(a) and (b) shows an example of an XML instance and the corresponding $XSketch$ model respectively. Based on [64], [66] extends *XSketch* to support queries with value predicates by introducing value distribution summaries.

The latest *XSketch*-based solution [65] studies the problem of estimating XML queries which project multiple nodes where all other existing approaches assume that the given queries only contain one projected node. The difference between them can be explained with the data in Figure 2.7.

Suppose we issue a branch query as shown in Figure 2.7(c). If this query only contains one projected node, for example, B, the query selectivity, which is directly

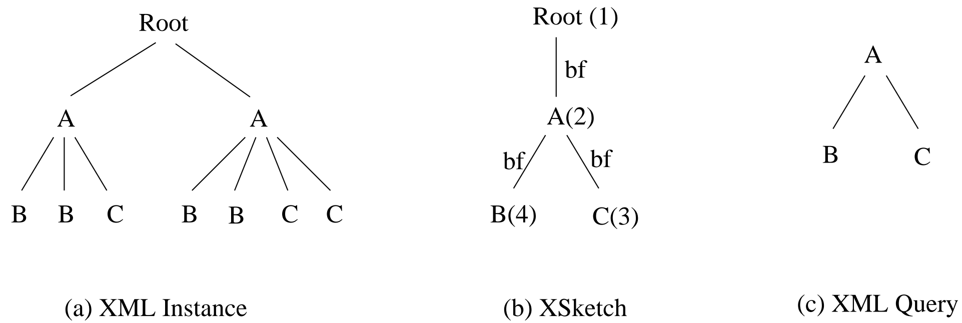


Figure 2.7: XML Instance, XSketch and XML Query

obtained from $XSketch$ (Figure 2.7(b)), is 4. On the other hand, if nodes B and C are both projected, the selectivity of this query is the number of all combinations of B and C if they occur under the same A nodes. Consequently, the selectivity should be 6 in this case. Obviously, the $XSketch$ model in Figure 2.7(b) cannot process this query.

To solve this problem, [65] records the additional distribution information of the outgoing edges of nodes, e.g., the distribution of B and C under A in Figure 2.7(a). Based on the edge distribution which is summarized in the edge histogram, [65] can estimate XML queries with multiple projected nodes.

However, the *XSketch* family suffers the problem of construction efficiency. [64] shows that building an optimal $XSketch$ model is an *NP*-hard problem. Hence, it utilizes a greedy refinement strategy to incrementally add complexity to the existing summary information. As a result, construction time grows quickly when the statistics increases. We describe details of this part in the experimental section of Chapter 5.

CHAPTER 3

A Path-Based Approach for Efficient Structural Join and Negation

3.1 Introduction

The support of structural join has become the key of efficient XML query processing. In this chapter, we design a novel path-based approach to expedite the structural join operation. In addition, the proposed approach can also efficiently process another important class of XML queries, negation. The underlying idea is to associate path information with each element node in an XML document so that we can filter out those nodes that clearly do not match the query, and identify a minimal set of nodes for the regular structural join between XML nodes. The unique features and contributions of the proposed approach are:

- A path-based labeling scheme is designed. It assigns a *path id* to each element to indicate the type of path on which this element node occurs. The scheme is compact, and the path ids have a much smaller size requirement compared to the node ids (see Section 3.6 on space requirement).
- The well-known node containment concept allows the structural relationship

between any two nodes in an XML document to be determined by their node labels. Here, we introduce the notion of *path id containment* and show how the path labeling scheme makes it easy to distinguish between parent-child and ancestor-descendant relationships.

- Based on the path id containment, a path join algorithm is proposed as a preprocessing step before regular node join to filter out irrelevant paths. The path join algorithm associates a set of relevant path ids to every node in the query pattern, thus identifying the candidate elements for the subsequent node join. Experimental results indicate that the relatively inexpensive path join can greatly reduce the number of elements involved in the node join.

- Different from other structural join index solutions, the proposed path join algorithm is easily extended to compute negation (i.e., with negated containment relationship in structural predicates). Experimental results demonstrate the effectiveness and efficiency of the proposed path-based solution in handling negation.

The rest of this chapter is organized as follows. Section 3.2 presents the path-based labeling scheme. Section 3.3 and Section 3.4 describes query evaluation on structural join and negation respectively. Section 3.5 and 3.6 present the experimental results. We conclude in Section 3.7.

3.2 Path-Based Labeling Scheme

In this section, we describe a path-based labeling scheme that assigns a *path id* to every element node in an XML document to indicate the type of path on which the node occurs. Each element node is now identified by a pair (path id, node id). The

node id can be given using any existing node labeling scheme, e.g., interval-based [57, 87], prefix [31], prime number [82]. Each text node is only labeled by a node id.

3.2.1 Path ID

This part introduces the structure, construction and maintenance of path id. Next, we discuss them respectively.

Structure

The path id is a sequence of bits. We first omit the text nodes from an XML document. Next, we find the distinct root-to-leaf paths in the XML document, considering only the tag names of the elements on the paths. We use an integer to encode each distinct root-to-leaf path in an XML document. The number of bits in the path id is given by the number of the distinct root-to-leaf element sequences of the tag names that occur in the XML document. Path ids are assigned to element nodes in a bottom-up manner as follows:

1. After omitting the text nodes in an XML document, let the number of distinct root-to-leaf paths in the XML document be k. Then the path id of an element node has k bits. These bits are initially set to 0.

2. The path id of every leaf element node is given by setting the *ith* bit (from the left) to 1, where i denotes the encoding of the root-to-leaf path on which the leaf node occurs.

3. The path id of every non-leaf element node is given by a bit-or operation on the path ids of all its element child nodes.

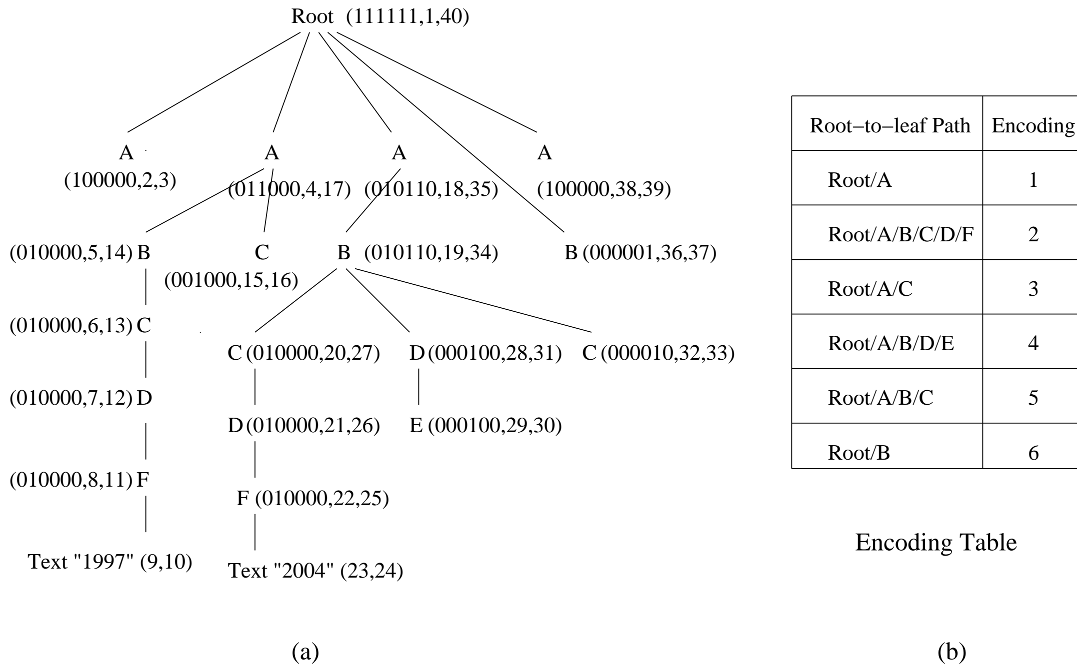


Figure 3.1: Path-Based Labeling Scheme

Example 3.1: Consider the XML instance in Figure 3.1(a) where the node ids have been assigned using interval based labeling scheme (the last two numbers). Figure 3.1(b) shows the integer encodings of each root-to-leaf path in the XML instance. Since there are six unique root-to-leaf paths, six bits are used for the path ids.

The path id of the element leaf node F (node id = (8,11)) is 010000 since the encoding of the path $Root/A/B/C/D/F$ on which F occurs is 2. The path id of the non-leaf node A (node id = (4,17)) is obtained by a bit-or operation on the path ids of its child nodes B and C, whose path ids are 010000 and 001000 respectively. Therefore, the path id of A is 011000. Finally, note that each text node is only labeled by a node id (interval). □

Construction

Path ids can be assigned to element nodes via a depth-first tree traversal on an

XML document. Algorithm 2 gives the details. The function $getPid()$ is called with the root node as parameter. The function initializes the path id of the node n to a sequence of 0 bits. If n is a leaf node and occurs on the root-to-leaf path with the integer encoding i, then the ith bit from the left is set to 1. Otherwise, the path id of n is the result of a bit-or operation on the path ids of all its child nodes. The path id of each child node is computed recursively by calling function $getPid()$.

Algorithm 2 Path ID Assignment

```
BitSequence getPid(Node n)
{

n.Pid = 000...;
if isElementLeaf(n) then
   n.Pid.ith = 1;
   /* i is the encoding of current root-to-leaf path*/
else
   child = n.firstElementChild();
   while child! = NULL do
      n.Pid = n.Pid | getPid(child);
      /* “ | ” denotes the bit-or operator*/
      child = child.nextElementSibling();
   end while
end if
return n.Pid

}
```

The encoding of a root-to-leaf path can be determined concurrently as the path ids are assigned. Since an XML document is typically stored in a strict-node-containment structure, a sequential scan of the XML file is functionally equivalent to a depth-first tree traversal. Considering Algorithm 1 that assigns node intervals by using the similar method, the collection of path information and node information can be carried out by a sequential scan of the XML data.

Maintenance

The problem of updating node ids can be solved by using dynamic node labeling schemes, such as [31], [82]. Here, we only discuss the maintenance of path ids: we explain by means of three examples.

Example 3.2: Case 1: The updates do not produce new root-to-leaf paths.

In this case, we do not need to relabel any existing path id since all paths remain unchanged. For example, in Figure 3.1, we can insert a new F node as the child of node D (node id = (7,12)) or delete node A (node id = (2,3)) without relabeling the path ids of other nodes. □

Example 3.3: Case 2: The updates produce a new root-to-leaf path and eliminate an existing path at the same time.

When some nodes are deleted from or inserted into XML data, new paths may be created to replace the original paths. To handle this situation, we can simply modify the corresponding paths in the encoding table. For instance, if F nodes (node id =(8,11), (22,25)) and their child nodes are deleted from the instance shown in Figure 3.1, all the paths $Root/A/B/C/D/F$ in XML data are converted to $Root/A/B/C/D$, which is a new path in the encoding table. As a result, we only need to change the path $Root/A/B/C/D/F$ (value = 2) in the encoding table to $Root/A/B/C/D$ since $Root/A/B/C/D$ totally replaces $Root/A/B/C/D/F$ in XML documents. □

Example 3.4: Case 3: The updates produce a new root-to-leaf path, and do not eliminate any existing path.

If this situation occurs, one more bit must be used to represent the new path, and the path ids of all nodes occurring in the new path must be relabeled. For example, the deletion of node F (node id = (8,11)) and its child node will produce

a new path $Root/A/B/C/D$, and a new bit (encoding = 7) is then utilized to represent this path. Therefore, the path ids of the ancestor nodes of F (node id = (8,11)) must be correspondingly modified. For instance, the updated path id of A (node id = (4,17)) should be 0010001. □

Storage

In order to facilitate the direct retrieval of elements with a specified path id, we design the following storage structure:

1. All the path ids of one element tag comprise the path id list of this element.
2. All the node ids of one element tag comprise the node id list of this element. In addition, all the node lists of XML elements are linked. For one element tag, the node list is first clustered by element path ids, and then sorted on the node ids.
3. Each path id in the path id list points to the first element with this path id.

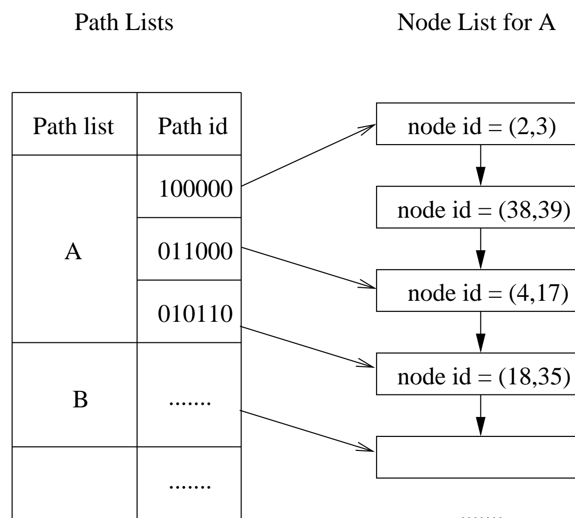


Figure 3.2: Storage Structure

Example 3.5: Figure 3.2 shows how the node labels of the sample XML document in Figure 3.1(a) are stored. Tag A has four occurrences with three distinct path ids. Thus the path list of element A contains three path ids, each of which points to the first element (of tag A) with the same path id. □

3.2.2 Containment of Path IDs

The well-known node containment concept states that the relationship between any two nodes in an XML document can be determined from their node labels. In this section, we introduce the notion of path id containment, which is based on our proposed path labeling scheme, and we examine the relationship of path id containment with node containment.

Definition 3.1 (Path ID Containment) *Let Pid_X and Pid_Y be the path ids of nodes with tags X and Y respectively. If all the bits with value 1 in Pid_X cover all the bits with value 1 at corresponding positions in Pid_Y, then Pid_X contains Pid_Y.*

The containment relationship between the path ids can simply be determined with a bit-and operation. That is, if $(Pid_X \;\&\; Pid_Y) = Pid_Y$ where & denotes the "bit-and" operation, then Pid_X contains Pid_Y.

Example 3.6: In Figure 3.1, the path id of A(010110,18,35) contains the path id of C(010000,20,27). □

Definition 3.2 (Strict Path ID Containment) *Let Pid_X and Pid_Y be the path ids of nodes with tags X and Y respectively. If Pid_X contains Pid_Y and $Pid_X \neq Pid_Y$, then Pid_X strictly contains Pid_Y.*

Example 3.7: In Figure 3.1, the path id of A(010110,18,35) also strictly contains the path id of C(010000,20,27). □

Next, we discuss the relationship between path id containment and node containment.

Theorem 3.1 *Let Pid_X and Pid_Y be the path ids of two nodes X and Y in an XML instance respectively. If node X is an ancestor of Y (or node X contains node Y), then Pid_X contains Pid_Y.*

Proof: *Based on the assignment of path ids, Pid_X must contain the path ids of all the child nodes of this X node since Pid_X is the result of the bit-or operation of all the child nodes. Similarly, the path ids of the child nodes of X must also contain the path ids of their own child nodes. This path id containment relationship continues until the node Y is reached. Thus Pid_X contains Pid_Y.* □

Example 3.8: In Figure 3.1, the node A(010110,18,35) is the ancestor of node C(010000,20,27); thus Pid_A 010110 contains Pid_C 010000. □

On the other hand, we can determine the node containment relationship based on the path ids of the nodes.

Theorem 3.2 *Let S_X be the set of element nodes labeled with X that have the same path id Pid_X, and let S_Y be the set of element nodes labeled with Y that have the same path id Pid_Y. If Pid_X strictly contains Pid_Y, then for each node $x \in S_X$, x must have at least one descendant y such that the path id of y contains Pid_Y.*

Proof: *Since Pid_X strictly contains Pid_Y, then all the bits with value 1 in Pid_Y must occur in Pid_X at the same positions. Further, Pid_X will have at least one bit with value 1 such that the corresponding bit (at the same position) in Pid_Y is 0. Consequently, elements x ($x \in S_X$) will occur in the same root-to-leaf paths as some elements y, such that the path ids of these y elements contain Pid_Y. As a result, all x elements must have some elements y as descendants.* □

Example 3.9: Consider again Figure 3.1. The path id 010110 for node A strictly contains the path id 010000 for node B. Therefore, each node A (node id = (18,35)) with path id 010110 must be the ancestor of at least one node B (node id = (19,34)) such that the path id (010110) of this B element contains Pid_B 010000. □

In the case where there are two sets of nodes with the same path ids, we need to check their corresponding root-to-leaf paths to determine their structural relationship. Suppose $Pid_X = Pid_Y$, it is obvious there exists at least one ancestor-descendant relationship between a node x, $x \in S_X$ and a node y, $y \in S_Y$. Recall that a path id, Pid, can be decomposed into a set of root-to-leaf paths, each of which corresponds to one bit with value 1 in Pid. Thus, the relationship of x and y can be determined by the relationship of their tags, X and Y, in any one of the component root-to-leaf paths of Pid_X (or Pid_Y).

Example 3.10: For instance, the nodes A and B (node ids (18,35) and (19,34)) in Figure 3.1 have the same path id 010110. We can decompose the path id 010110 into three root-to-leaf paths with the encodings 2, 4 and 5 since the bits in the corresponding positions are 1. Thus, by looking up any of these paths (in the encoding table) where nodes A and B occur, we know that all the nodes A with path id 010110 have B descendants with this path id. □

In addition, the *encoding table* can also help determine the exact relationship (parent-child or ancestor-descendant) between nodes by checking the positions of tags in the root-to-leaf paths. Note that only one of root-to-leaf paths is needed since the relationship of the two element tags will be the same in all of them.

Example 3.11: Consider Example 3.10 again. By checking the encoding table, we can further know that all the nodes A with path id 010110 are parents of B with this path id since in the corresponding root-to-leaf path, tag A occurs immediately

before tag B. □

In summary, if Pid_X and Pid_Y are the path ids of two sets of nodes X and Y respectively, we can determine the exact relationship (parent-child, grandparent-grandchild...) between these two sets of nodes from the encoding table, provided that a tag name occurs no more than once in any path (e.g., non-recursive XML elements). For example, suppose nodes X and Y have the same path ids, and their corresponding root-to-leaf path is "$X/Y/X/Y$". In this case, the structural relationship between X and Y can only be determined with an examination of their node labels (node ids).

3.3 Query Evaluation of Structural Join

In this section, we give the details of the proposed path-based solution for evaluating XML structural join. Structural join XML queries evaluate the containment relationship between nodes. There are two steps in the proposed solution: (1) path join, and (2) node join. The algorithms for carrying out these two steps are called $PJoin$ and $NJoin$ respectively.

3.3.1 $PJoin$

The $PJoin$ algorithm (Algorithm 3) aims to eliminate as many unnecessary path types as possible, thus minimizing the elements involved in the subsequent $NJoin$.

Given an XML query modeled using a tree structure T, $PJoin$ retrieves the set of path ids for every element node in T. Starting from each element leaf nodes in T, $PJoin$ performs a binary path join between each pair of parent-child nodes. This process is carried out in a bottom-up manner until the root node is reached. Finally, a top-down binary path join is performed to further remove unnecessary

Algorithm 3 *PJoin* (T)

Input: T - An XML Query.
Output: Path ids for the nodes in T.

1. Associate every node in T with its path ids.
2. Perform a bottom-up binary path join on T.
3. Perform a top-down binary path join on T.

path ids.

A binary path join takes as input two lists of path ids, one for the parent node and the other for the child node. A nested loop is used to find the matching pairs of path ids based on the path id containment property. Any path id that does not satisfy the path id containment relationship is removed from the lists of path ids of both the parent node and the child nodes.

Example 3.12: Consider the XML query $T1$ in Figure 3.3(a) where the lists of path ids have been associated with the corresponding nodes. For simplicity, we assume that the path ids with the same subscripts satisfy the path id containment relationship, that is, b_1 contains c_1, b_3 contains d_3 and e_3, etc. The *PJoin* algorithm evaluates the query $T1$ by first joining the path ids of node B with the path ids of node C. The path ids c_1 and c_3 are contained in the path ids b_1 and b_3 respectively. Thus, we remove b_2 and b_4 from the set of path ids of B.

Next, the algorithm joins the set of path ids of D with that of E. This is followed by a join between the sets of path ids of B and D. The result of the bottom-up path join is shown in Figure 3.3(b).

Finally, the algorithm carries out a top-down path join on $T1$, starting from the root node B. Figure 3.3(c) shows the final sets of path ids that are associated with each node in $T1$. Compared to the initial set of path ids associated with each node in Figure 3.3(a), the *PJoin* algorithm has greatly reduced the number of elements

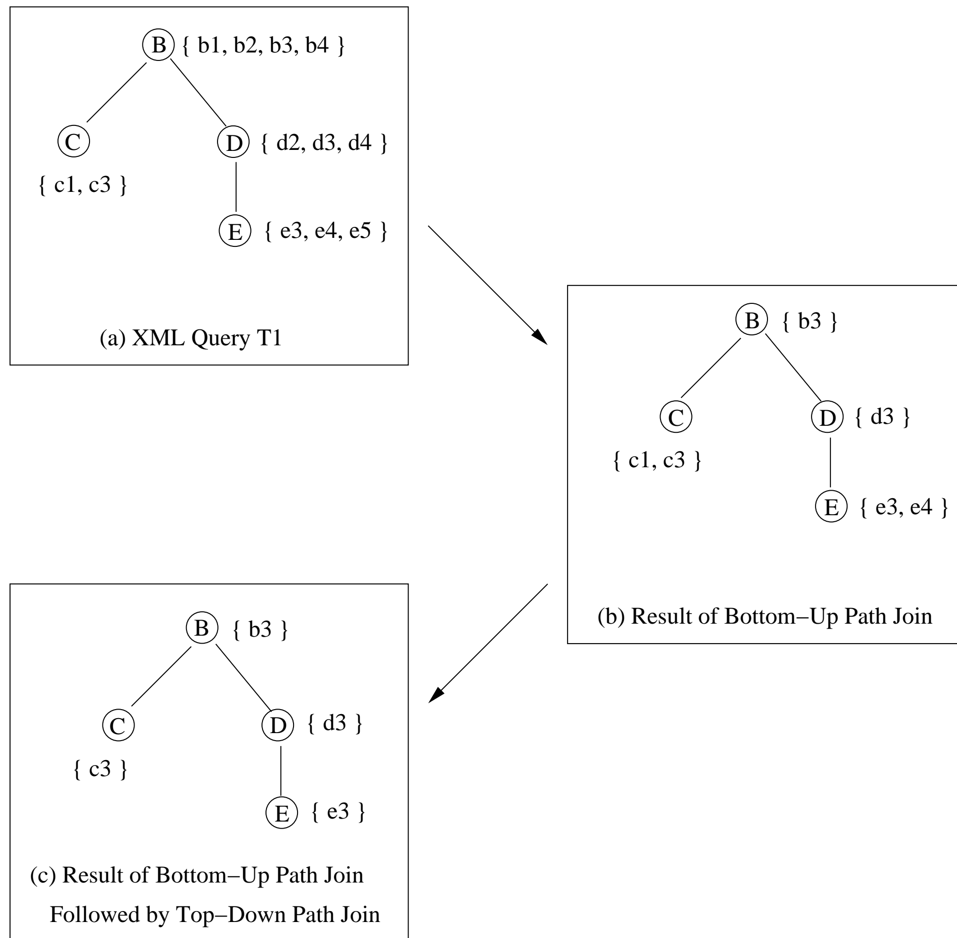


Figure 3.3: Example of $PJoin$

involved in the query. The subsequent node join is now almost optimal.

Note that omitting either the bottom-up or top-down tree traversal does not produce this optimal result. This is because a single tree traversal cannot project the result of each binary path join to the nodes which have been processed earlier. In Figure 3.3(b), elements C and E contain unnecessary path ids $c1$ and $e4$ compared with final optimal results. □

Optimization of PJoin

We observe that the $PJoin$ algorithm can be optimized such that only the set of

path ids for the parent node is updated during the bottom-up binary join, and only the set of path ids for the child node is updated during the top-down binary join. This is because a two-phase tree traversal is sufficient to propagate the changes in the sets of path ids involved in every binary path join into the final path ids associated with all the nodes in a query. Concretely speaking, in the bottom-up path join, the path ids of each node must contain all the path ids of child nodes, and during the procedure of top-down join, the path ids of each node are contained by its parent node (ancestor nodes). As a result, the final path id set associated with each node must be the optimal result set.

Example 3.13: With this optimization, the path ids of D and E in Figure 3.3(b) become $\{d_3, d_4\}$ and $\{e_3, e_4, e_5\}$ respectively, since we only update the path ids of the parent nodes when joining D and E, B and D. However, the final sets of path ids for the nodes in the query remain unchanged (see Figure 3.3(c)). □

3.3.2 *NJoin*

The output of the *PJoin* algorithm is a set of path ids for the element nodes in a given query tree. Elements with these path ids are retrieved for a node join to obtain the result of the query. Algorithm 4 shows the details of *NJoin*.

The node join can be performed by using the existing holistic join methods TwigStack [19] or iTwigJoin [24]. That is, the element nodes are retrieved according to the path ids obtained from *PJoin* while all the value (text) nodes are retrieved directly. Finally, a holistic structural join is carried out on all the lists obtained.

We observe that in structural join solution [19], TwigStack algorithm requires that the input stream for every node in the query must be an ordered list of node ids. However, Line 1 of Algorithm 4 produces a set of ordered sublists, each of which is associated with a path id obtained in the *PJoin*. Therefore, when performing

Algorithm 4 $NJoin(T)$

Input: T - An XML Query.
Output: All occurrences of nodes in T.

1. Retrieve the elements according to the path ids associated with the nodes in T
2. Retrieve the values imposed on the element nodes.
3. Perform holistic structural join on T.

structural join by using TwigStack [19], we need to examine these multiple sublists of node ids for an element tag to find the smallest node id to be processed next. However, if iTwigJoin solution [24] is adopted, we do not need to merge the sublists of elements purposely since iTwigJoin [24] is specially designed for multiple streams.

3.3.3 Discussion

Path join is designed to reduce the number of elements involved in the subsequent node join. In this part, we analyze the effectiveness of the proposed path join.

Definition 3.3 (Exact Pid Set) *Let P be a set of path ids obtained for a node N in an XML query T after path join. P is an* Exact Pid Set *with respect to T and N if the following conditions hold:*

1. *for each path id $p_i \in P$, the element with tag N and path id p_i is a result for T, and*
2. *for each path id $p_j \notin P$, the element with tag N and path id p_j is not a result for T.*

Definition 3.4 (Super Pid Set) *Let P be a set of path ids obtained for a node N in an XML query T after path join. P is a* Super Pid Set *with respect to T and*

N if each element with a tag N in the final result (after node join) is associated with a path id p_i such that $p_i \in P$.

Clearly, each element node is associated with its super *Pid* set after path join. The result is optimal when these super *Pid* sets are also the exact *Pid* sets.

Next, we examine the situations where path join yields exact *Pid* sets. We assume that the XML elements are non-recursive.

Simple Path Queries

Suppose query T is a simple path query without value predicates. Then after path join, each node in T has an exact *Pid* set associated. This is because all path ids that satisfy the path id containment property are reserved in the adjacent nodes of T. Since this containment property is transitive, all path ids of a node N in T contain the path ids of its descendant nodes, and vice versa. Moreover, the encoding table for the paths can identify the exact containment relationship (parent-child or ancestor-descendant) between the nodes in T. Therefore, given a simple path XML query without value predicates, path join eliminates all elements (path ids) that do not appear in the final result sets.

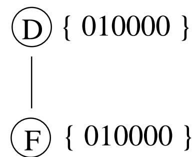


Figure 3.4: Example of Exact Pid Set

Example 3.14: Consider the query in Figure 3.4, which is issued on the XML instance in Figure 3.1. After path join, all the nodes with path ids remaining (node ids of D =(7,12) (21,26), node ids of F = (8,11) (22,25)) are the results for the query. □

Branch Queries

If a query T is a branch query, then we cannot guarantee that the nodes on the branch path have exact Pid sets. This is due to the manner in which path ids are assigned to elements. In other words, the path id is designed to capture the containment relationship, but not the relationship between sibling nodes.

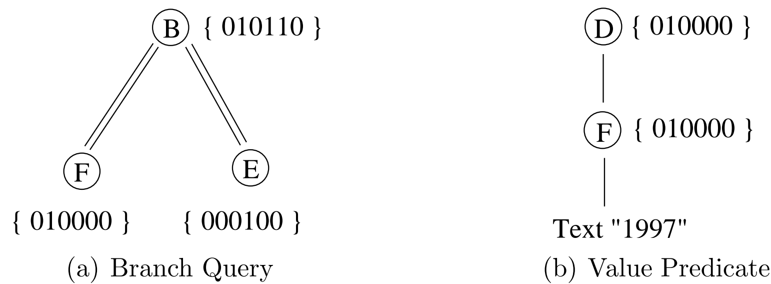


Figure 3.5: Examples of Super Pid Set

Example 3.15: Consider the query in Figure 3.5(a), which is issued on the XML instance in Figure 3.1. After path join, node F is associated with path id 010000. However, we see that only F(010000,22,25) is an answer to the query while F(010000,8,11) is not. This is because we can only detect that 010000 (path id of F) is contained by 010110 (path id of B), but do not know whether an F element with path id 010000 has sibling E. Finally, note that the path id set of B in Figure 3.5(a) is guaranteed to be an exact Pid set since B has no sibling nodes.

□

Queries with Value Predicates

The path-based labeling scheme only assigns path ids to element nodes, but not to value (text) nodes. Therefore, given an XML query with value predicates, the element nodes with the matching path ids (after path join) can only satisfy the structural relationship, but not the value constraints. As a result, the path id set

of each node in the query pattern may not be the exact Pid set.

Example 3.16: Figure 3.5(b) shows an XML query with value predicates that will lead to a super Pid set after path join. The nodes D (010000,21,26) and F (010000,22,25) in Figure 3.1 are not answers to the query although the path id 010000 occurs in the path id sets of D and F respectively after path join. □

In summary, for non-recursive XML data, exact Pid sets are associated with element nodes in simple XML query patterns after path join. Although super Pid sets are attached to the elements involved in branch queries and queries with value predicates, these super Pid sets are much fewer than the full path id sets of elements. In other words, path join still remains efficient in eliminating unnecessary path ids. This is clearly shown in our experimental section.

3.4 Query Evaluation of Negation

The evaluation of negation is quite different from the evaluation of structural join. In this section, we first define a model called XQuery tree to model queries involving negated containment relationships. Then we will present the corresponding path join and node join algorithms, $PJoin^+$ and $NJoin^+$, for such queries.

3.4.1 XQuery Tree

XML queries are typically viewed as tree patterns. To model queries involving negation, we augment the query pattern tree with two new features: *node projection* and *not* operator. We call the augmented query pattern tree the *XQuery Tree*.

Definition 3.5 (XQuery Tree) *An XQuery Tree is defined as follows:*

1. *It is a tree $T = (V, E)$ where V and E denote the set of nodes and edges*

respectively.

2. *The single line edge and double line edge denote a parent-child relationship and an ancestor-descendant relationship respectively.*

3. *The nodes to be projected are circled.*

4. *A negated containment relationship between two nodes is specified by putting the symbol "!" next to the edge. We call such an edge a negated edge.*

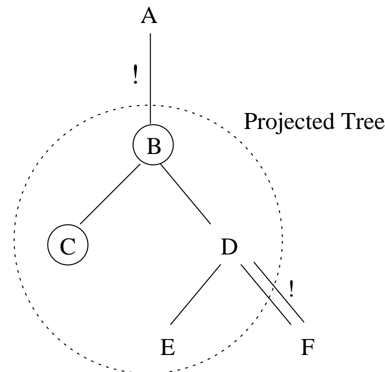


Figure 3.6: XQuery Tree.

Example 3.17: Figure 3.6 shows an example of the XQuery tree. It specifies a query that retrieves all the matching occurrences of elements B and C such that B is not contained in A and B has child nodes C and D while D has a child node E but does not have a descendant node F. Formally, this query can be represented by using XQuery as follows:

$$
\begin{aligned}
&For\ \ \$v\ \ In\ \ //B \\
&Where\ \ exists(\$v/C)\ \ and\ \ exists(\$v/D/E)\ \ and \\
&\qquad count(A/\$v) = 0\ \ and\ \ count(\$v/D//F) = 0 \\
&Return\ \ \{\$v\}\ \ \{\$v/C\}
\end{aligned}
$$

□

In this work, we assume that negated edges do not occur between the projected nodes of a query. This is because such queries are typically meaningless, e.g., retrieve all the elements A and B such that A does not contain B. Based on this assumption, we can deduce that given an XQuery tree T, there exists some subtree T' of T such that T' contains all the projected nodes in T and all edges in T' are not negated edges.

Definition 3.6 (Projected Tree T_P) *Let $T = (V, E)$ be an XQuery tree, and S be the set of subtrees $T' = (V', E')$ of T, such that*

1. $V' \subseteq V$ *and*
2. V' *contains all the projected nodes in T, and*
3. *for any $e \in E'$, e is not a negated edge.*

The largest T' in S is defined as the projected tree T_p of T.

Example 3.18: For instance, the projected tree of the XQuery tree in Figure 3.6 is shown within the dashed circle. □

Given an XQuery tree T, we define the subtree above T_P as tree T_P^a and the subtree below T_P as T_P^b respectively.

Definition 3.7 (Tree T_P^a) *Given an XQuery tree T, let R be the root node of T_P, and e be the incoming edge of R. We define T_P^a as the subtree obtained from T - T_R - e, where T_R denotes the subtree rooted at R.*

Definition 3.8 (Tree T_P^b) *Given an XQuery tree T, we define T_P^b as the subtree rooted at C, where C denotes a child node of the leaf nodes of T_P.*

Example 3.19: Consider Figure 3.6 again. The nodes A and F form the trees T_P^a and T_P^b of T respectively. Note that an XQuery tree T has at most one T_P^a and possibly multiple T_P^b. □

From the above definitions, a tree T_P^a or T_P^b may contain negated edges. However, queries with negated edges in T_P^a or T_P^b may have multiple interpretations. For example, the query "A does not contain B, and B does not contain C, where C is the projected node" has different semantics depending on the applications. In this work, we focus on queries whose subtrees T_P^a and T_P^b do not contain any negated edges.

3.4.2 *PJoin*$^+$

While the *PJoin* algorithm can efficiently filter out unnecessary path ids for queries only involving structural join, it does not work well for queries involving negation.

Example 3.20: Figure 3.7(a) shows a query $T2$ that involves a negated containment relationship between nodes B and C. The optimal sets of path ids for the nodes in the query are given in Figure 3.7(c). However, the *PJoin* algorithm yields the sets of path ids as shown in Figure 3.7(b). It can be observed that the path ids c_2, d_2 and e_2 are not eliminated for the projected nodes C, D and E. □

Now let us examine how the *PJoin* algorithm processes the query in Figure 3.7(a). We denote the sets of path ids for nodes B and C obtained after Step i as P_b^i and P_c^i respectively:

Step 1. Obtain the set of path ids for node C (P_c^1) after the bottom-up binary path join between C and D, and C and E. That is, $P_c^1 = \{c_2, c_3, c_4\}$.

Step 2. Remove the path ids of node B which do not contain any element in P_c^1. We have $P_b^2 = \{b_1\}$

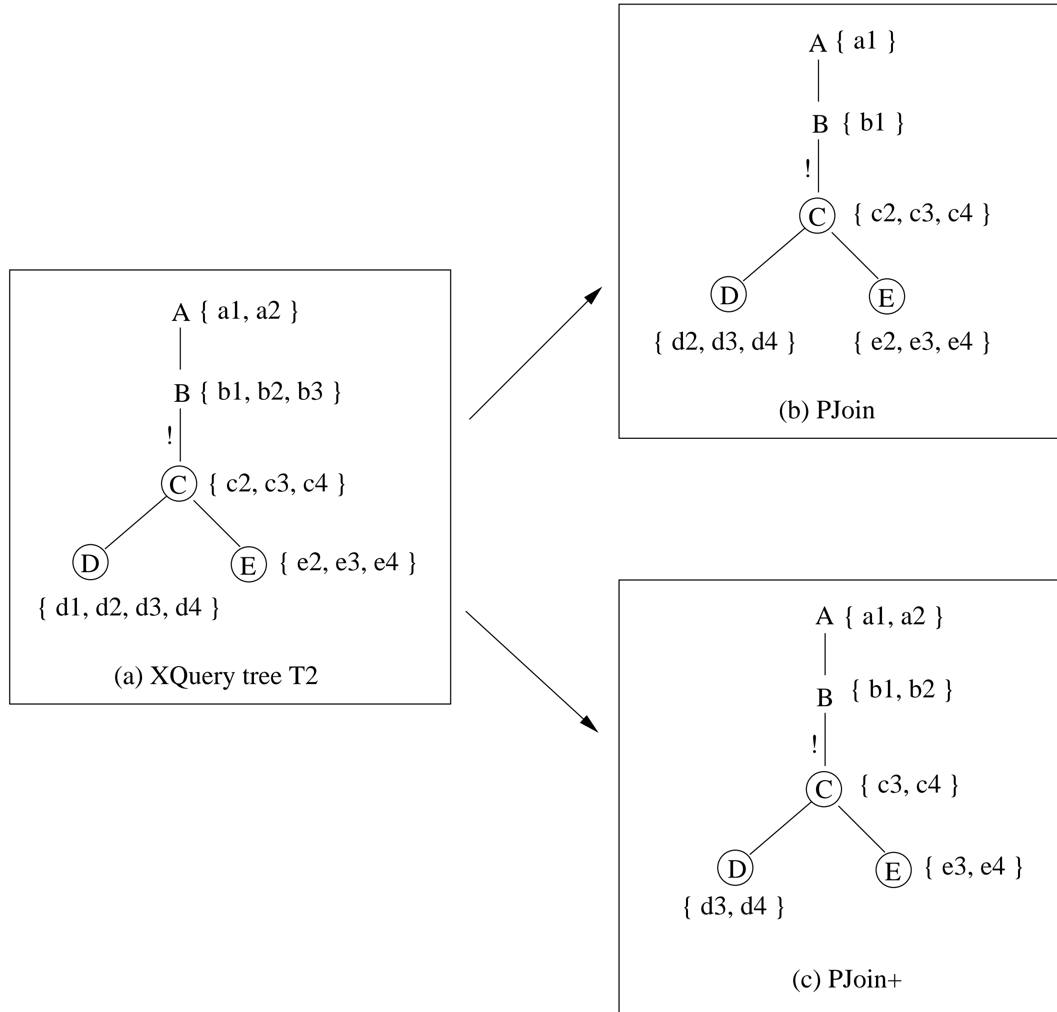


Figure 3.7: Example of $PJoin^+$

Step 3. Update P_b^2 by performing a binary path join between A and B. Then $P_b^3 = \{b_1\}$

Step 4. Remove the path ids of C which are not contained by any element in P_b^3. We have $P_c^4 = \{c_2, c_3, c_4\}$

Step 5. Update the path ids of nodes in the subtree rooted at C using the top-down path join.

P_b^3 is a subset of P_b^2 since the update in Step 3 is a reduction operation. There-

fore, P_c^4 must be exactly the same as P_c^2 (P_c^1). This is because all the elements in P_c^2 are not contained by any path id in P_b^2, and thus they cannot be contained by any path id in any subset of P_b^2, which is P_b^4 or P_b^3. In fact, the constraint that is imposed on nodes A and B in Step 3 does not apply to the entire query.

We observe that the proper way to evaluate a negated containment relationship between path ids is to only update the path ids of the nodes in the projected tree. This leads to the $PJoin^+$ algorithm.

The basic idea behind $PJoin^+$ (Algorithm 5) is that given a query T, we first apply $PJoin$ on T_P^a and T_P^b respectively. Then the path ids of the leaf node of T_P^a and the root node of T_P^b are used to filter out the path ids of the corresponding nodes in T_P.

Algorithm 5 $PJoin^+(T)$

Input: T - An XQuery-tree
Output: Path ids for the nodes in T.

1. Associate every node in T with its path ids.
2. Perform $PJoin$ on T_P^b and T_P^a.
3. Perform a path anti-join between the root node(s) of T_P^b and their parent node(s) if necessary.
4. Perform a bottom-up binary path join on T_P.
5. Perform a path anti-join between the leaf node of T_P^a with its child node if necessary.
6. Perform a top-down binary path join on T_P.

The input to Algorithm 5 is an XQuery tree T with a set of projected nodes. We first determine the projected tree T_P of T. After that, the $PJoin$ algorithm is carried out on T_P^b and T_P^a (if any) respectively. Next, $PJoin$ (Lines 4 and 6) is performed on T_P, but the path ids of the root node(s) of T_P^b and the leaf node in T_P^a are utilized to filter out the path ids of their parent node(s) and child node

respectively.

Note that if the set of path ids for the root node (leaf node) of T_P^b (T_P^a) is a super Pid set but not an exact Pid set with respect to T_P^b (T_P^a), then the path anti-join operation in Lines 3 (5) of Algorithm 5 should be skipped. This is because the super Pid set of the root node (leaf node) of T_P^b (T_P^a) could erroneously remove path ids from its parent node (child node), and we may miss some correct answer in the final query results.

Example 3.21: Consider again query $T2$ in Figure 3.7. The projected tree is the subtree rooted at node C. A $PJoin$ is first performed on tree T_P^a which contains nodes A and B. The set of path ids for B obtained is $\{b_1, b_2\}$. Next, bottom-up path join is carried out on T_P. Since T_P^a is a simple path query without value predicates, the path id set associated with B is the exact Pid set. Then we can perform a path anti-join between nodes B and C. This step eliminates c_2 from the path id set of C. Finally, a top-down path join is performed on T_P, which eliminates d_1 and d_2 from the set of path ids for D, and e_2 from the set of path ids for E. □

It can be observed that the $PJoin^+$ algorithm is reduced to the $PJoin$ algorithm when the XML query does not contain any negation. This is because lines 2, 3 and 5 in Algorithm 5 will not be executed and T_P is the same as T.

3.4.3 $NJoin^+$

Algorithm 6 shows the details of corresponding $NJoin^+$ method. [86] proposes a holistic twig join method, namely TwigStackList¬, to evaluate negation. However, TwigStackList¬ [86] requires that the tree T_P^a must be empty since it only guarantees that each node in T_P satisfies the subtree rooted at this node when evaluating the query. Thus in Algorithm 6, if T_P^a is null, we then directly carry out

TwigStackList¬ [86] to calculate the final results. Otherwise, we first evaluate the trees T_P^a, T_P^b and T_P respectively, and then merge these intermediate results.

Algorithm 6 $NJoin^+(T)$

Input: T - An XQuery tree
Output: All occurrences of nodes in T_P

- If T_P^a is null, perform TwigStackList¬ on XQuery tree T.
- Otherwise, perform $NJoin$ on T_P^a, T_P^b and T_P, then merge the intermediate results.

3.5 Experiments - Part 1

The performance study in this chapter consists of two sections. We first evaluate the existing index structures for XML structural join. Then we examine the performance of our proposed path-based approach.

In this section, we implement the B^+-tree [28], XR-tree [43] and XB-tree [19] structural join algorithms in Java. We also implement a variant of the XR-tree that decreases update cost, and call it XR-v. In the original XR-tree [43], all elements that are stabbed by the keys in one index node are stored in the stab list of the index node. Since the stab list is an ordered element list, the cost to maintain a large stab list is high. In XR-v, we store elements that are stabbed by different keys in separate lists. Although XR-v may increase query cost compared to the original XR-tree, it is able to decrease update cost when the stab list is large.

Datasets and Query Workload

To control the characteristics of the dataset, we generate synthetic XML data. The synthetic XML data set only contains two element tags, "*Ancestor*" and "*Descendant*". Both element tags are recursive elements. In addition, we fix

the number of "*Ancestor*" and "*Descendant*" elements at 240,000 and 480,000 respectively, and vary the selectivity values of both nodes and levels of nestings. Here, the selectivity values denote the percentage of number of matching elements to the total number of element. Table 3.1 shows the characteristics of the dataset generated, and the ranges of values for the parameters. A structural join is then carried out on the elements "*Ancestor*" and "*Descendant*" ($A//D$) to test the query evaluation performance. All the experiments are carried out on a Pentium IV 800 MHz CPU with 1 GB RAM. The page size is 4 KB and we record the average results of five runs.

Element	Number	Node Selectivity	Nesting
Ancestor	240,000	0.1%-70%	1-220
Descendant	480,000	1%-80%	1-140

Table 3.1: Dataset Characteristics of Experiments 1

3.5.1 Query Evaluation Performance

This set of experiments evaluates the query performance of the various indices. The I/O cost incurred by structural join is used as the performance metric. All index structures are built by bulkloading the elements. Node occupancy, except for the root node, is kept at 50%.

Ancestor Selectivity

We first investigate the effect of low ancestor selectivity. We limit ancestor selectivity to the range of 0.1%-0.5% while the ancestor and descendent nesting levels, and descendant selectivity are fixed at 50, 5 and 10% respectively. Figure 3.8(a) shows the results when we keep the root nodes of the various indices in the buffer while Figure 3.8(b) presents the results when a 100 KB buffer is used. The XB-tree gives

the best performance in both situations by avoiding the sequential scans needed in the B^+-tree.

In Figure 3.8(a), the XR-tree and its variant shows the worst performance as they need to access the stab lists multiple times. The XR-tree (XR-v) benefits the most from an increasing buffer size. With sufficient buffer, we can pin the stab list pages in the buffer, and the performance of the XR-tree (XR-v) improves dramatically (see Figure 3.8(b)).

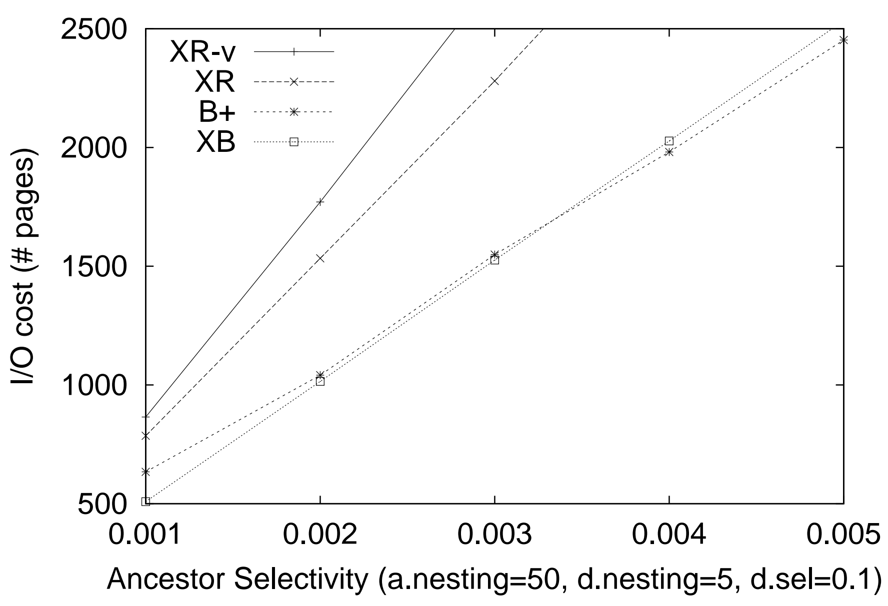
Figure 3.9 shows the results for higher values of ancestor selectivity: 10%-70%. Since the "*Ancestor*" elements involved in the join are uniformly distributed, we need to access most of the leaf pages in the index structures. Thus, it is not surprising the I/O costs for both the XB-tree and the B^+-tree are almost the same.

When buffer size is increased, and ancestor selectivity is above 40%, the XR-tree and XR-v outperforms the XB-tree and the B^+-tree (see Figure 3.9(b)). Unlike the B^+-tree and the XB-tree which need to access most of the leaf pages, the XR-tree and the XR-v only access the leaf page that contains the last ancestor element that covers the descendant element. The rest of the ancestor elements that contain the descendant element can be obtained from the stab lists which are most likely to be in the buffer.

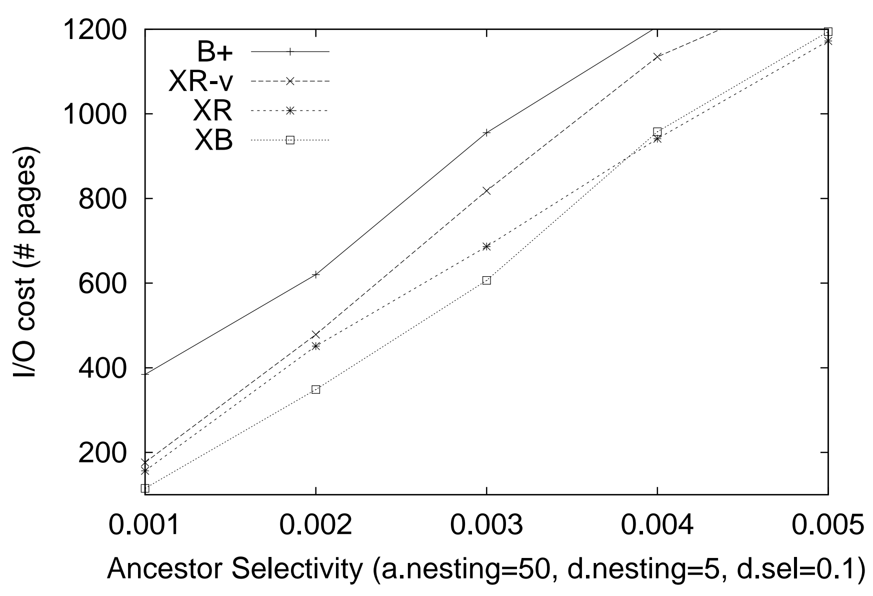
Descendant Selectivity

Next, we examine the effect of varying descendant selectivity. The results are shown in Figure 3.10. Again, the performance of the XR-tree and the XR-v largely depends on the buffer size for the same reasons given above.

I/O cost does not increase when descendant selectivity is above 10% for all index structures in both graphs of Figure 3.10. The "*Descendant*" elements involved in the join are uniformly distributed in the entire "*Descendant*" element set. Hence,

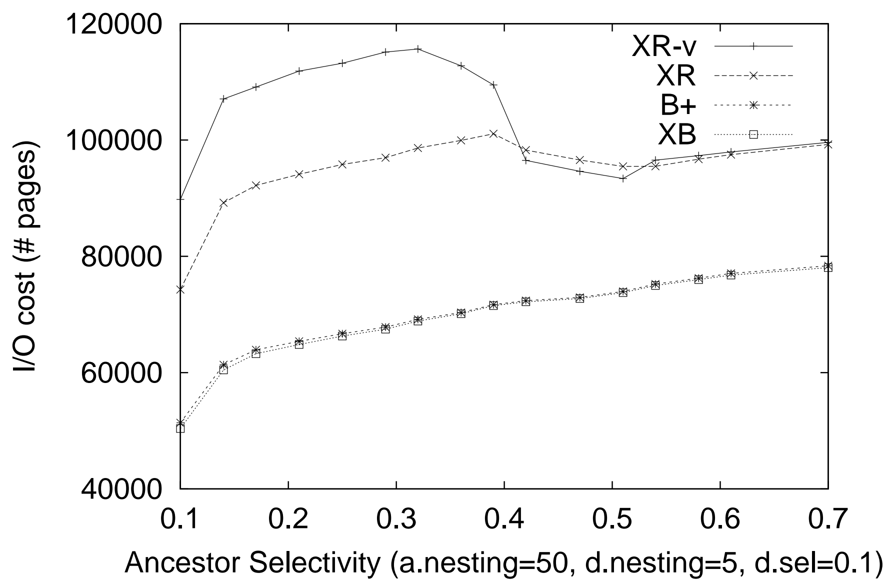


(a) Root in Buffer Only

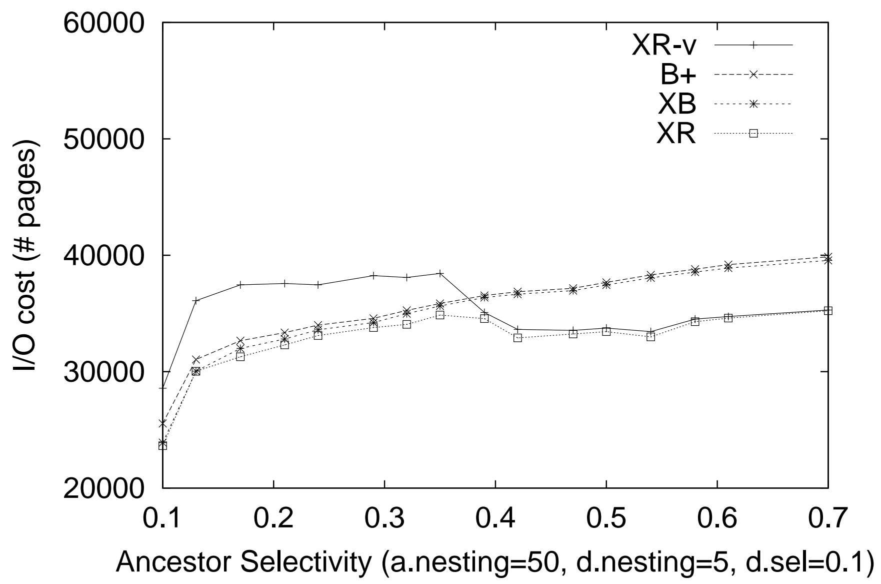


(b) Buffer Size = 100K

Figure 3.8: Low Ancestor Selectivity

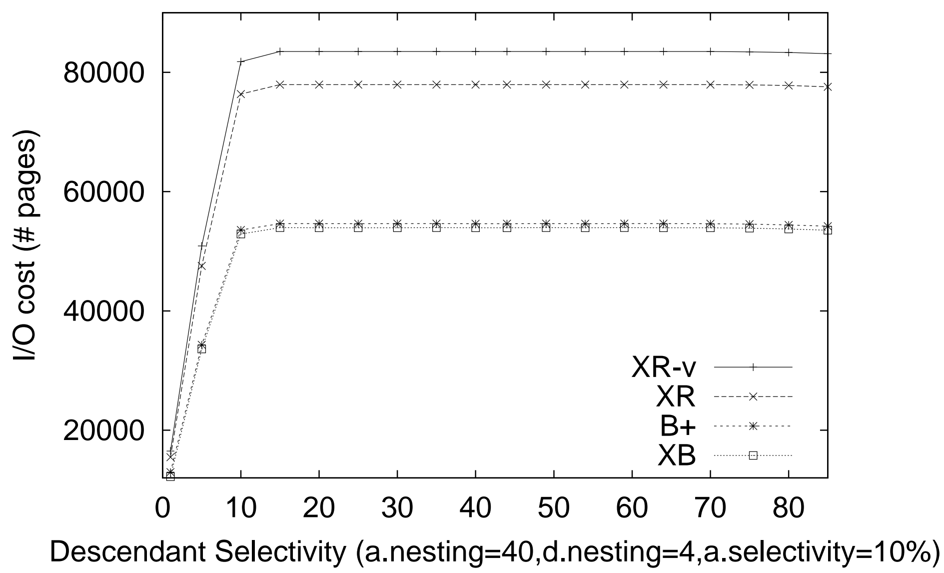


(a) Root in Buffer Only

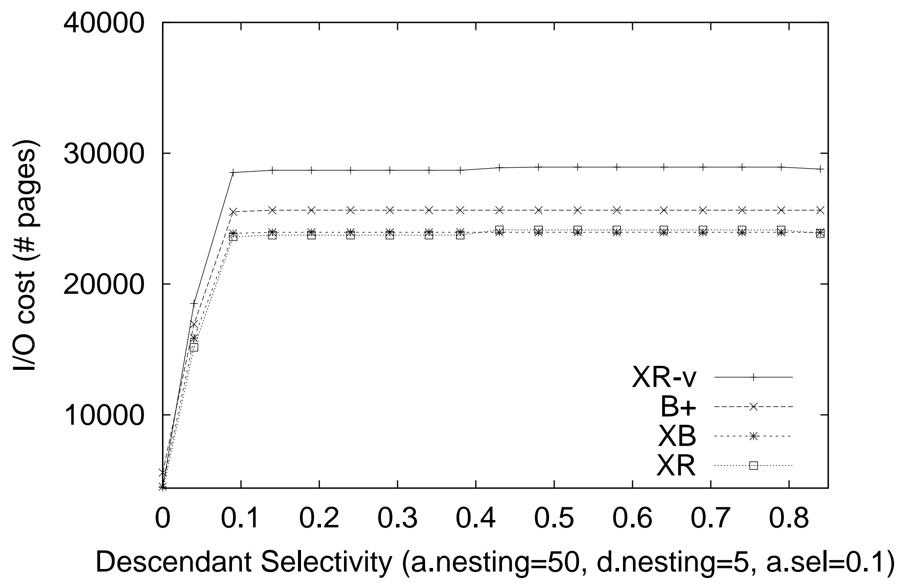


(b) Buffer Size = 100K

Figure 3.9: High Ancestor Selectivity

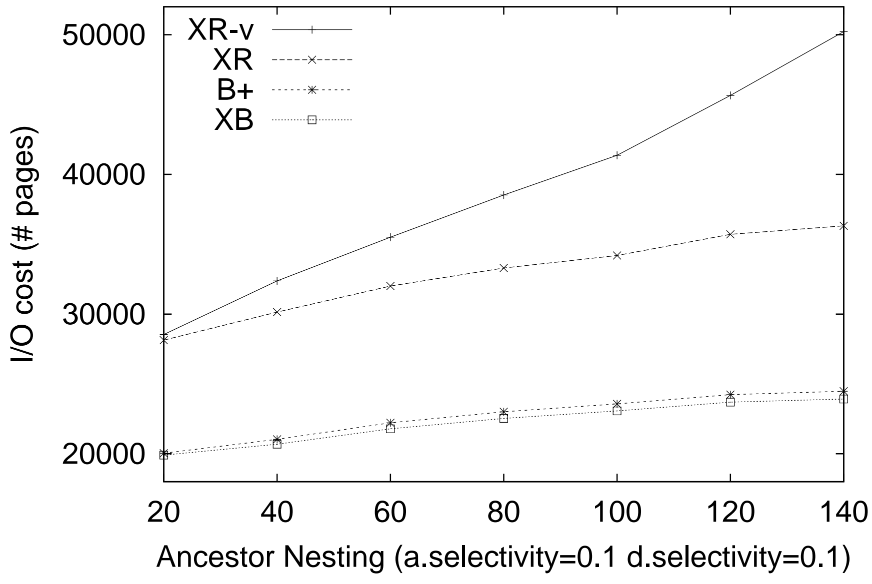


(a) Root in Buffer Only

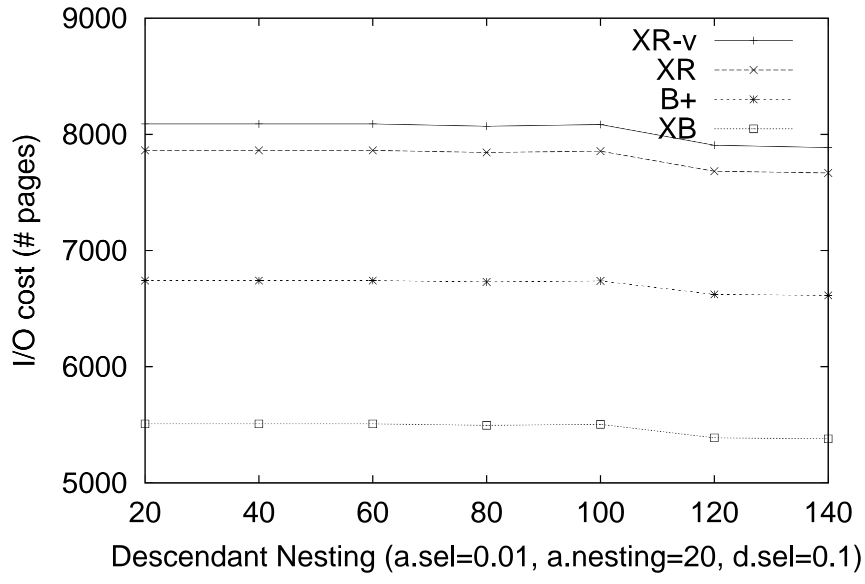


(b) Buffer Size = 100K

Figure 3.10: Descendant Selectivity



(a) Ancestor



(b) Descendant

Figure 3.11: Levels of Nestings

the increasing number of elements involved in the join would not lead to additional I/O costs.

Nesting Levels

In this experiment, we examine the effect of varying levels of ancestor and descendant nesting. Only the root nodes of the index structures are kept in the buffer. Figure 3.11(a) shows that increasing the levels of ancestor nesting increases the size of the stab lists and contributes to the rapid performance deterioration in the XR-tree (and the XR-v). In contrast, the curves for the B^+-tree and the XB-tree are almost flat. Figure 3.11(b) indicates that the various index structures are independent of the levels of descendant nesting.

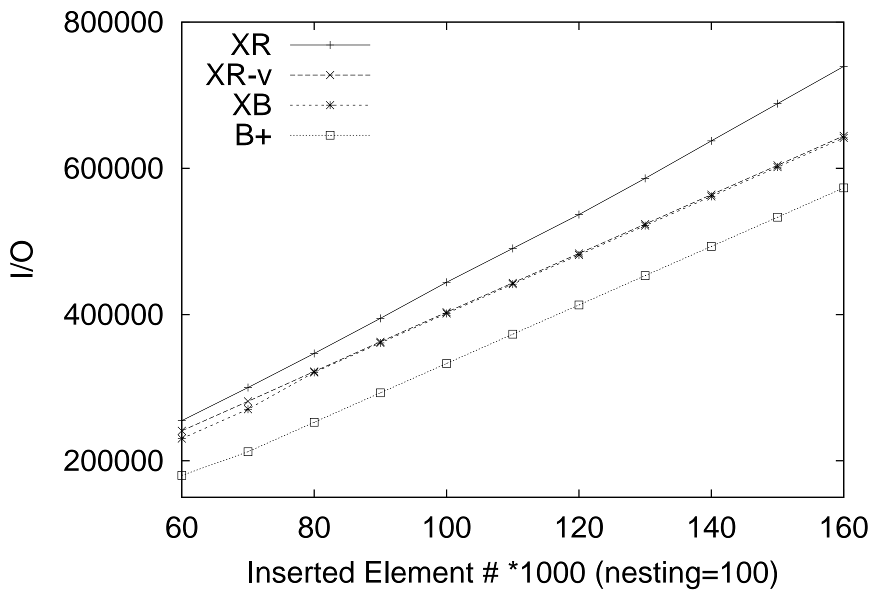
3.5.2 Update Performance

This set of experiments examines insertion and deletion costs. The buffer is turned off here. The index structures are initially empty, and we randomly insert 160,000 element intervals. Figure 3.12(a) shows the insertion results. The XR-tree has the highest I/O cost since it has to maintain a set of ordered stab lists. Compared to the XR-tree, XR-v performs better due to its smaller individual stab lists.

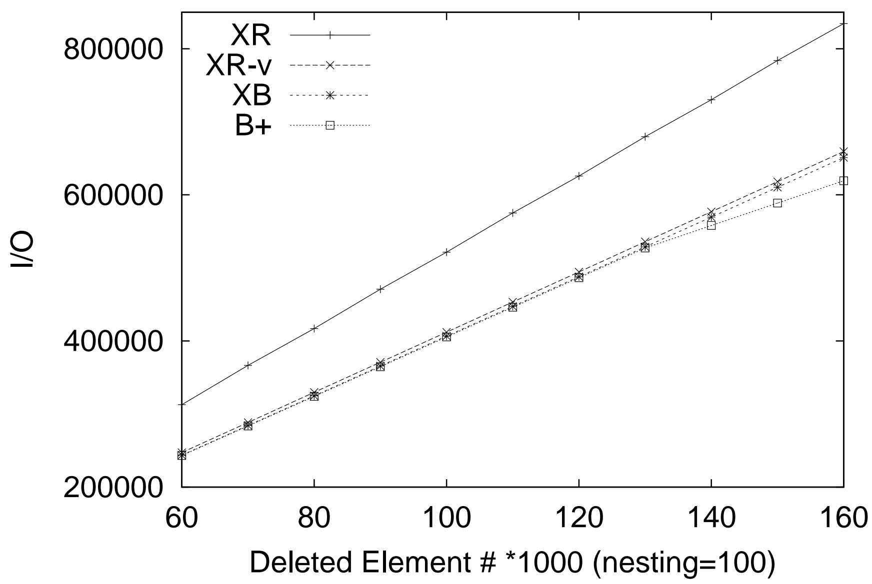
Next, we randomly delete 160,000 elements from the index trees and record the number of I/Os. Figure 3.12(b) shows the deletion cost. The performance of deletion comes very near to that of insertion. Considering the characteristics of the different index structures, the results of the update experiments are as expected.

3.5.3 Space Utilization

Finally, we investigate the space consumption of the various index structures. Each index is built by bulkloading 240,000 elements, and every node in the index is 50%



(a) Insertion



(b) Deletion

Figure 3.12: Update Cost

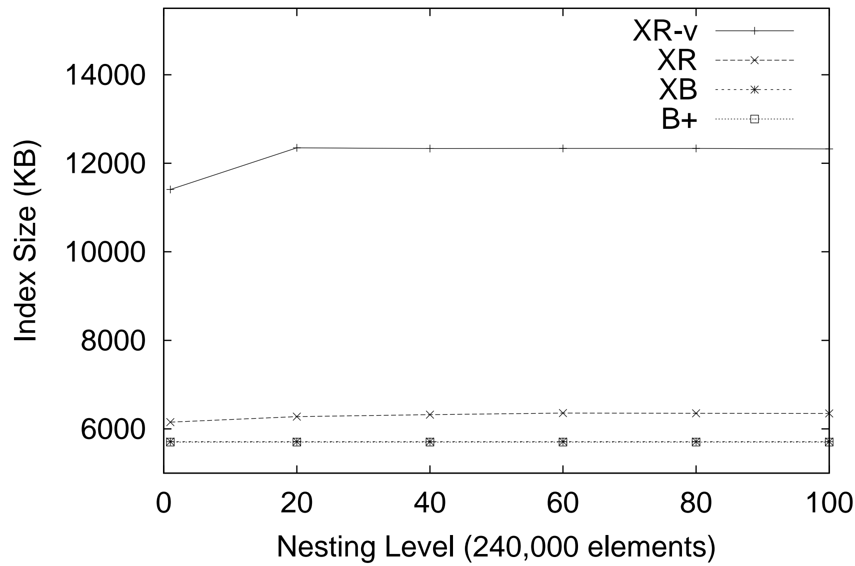


Figure 3.13: Space Consumption

full except for the root node. We only vary the number of nesting levels since this is the only parameter that may affect space utilization. Figure 3.13 indicates that the XR-v requires the most storage space due to its individual stab lists. As the number of nesting levels increases, the space consumption of the XR-tree and XR-v increase slightly while the sizes of the XB-tree and B^+-tree remain stable.

3.5.4 Summary

In this part, we have compared and analyzed the performance of the B^+-tree, the XB-tree and the XR-tree for XML structural join. The experiment results indicate that all three indexes give comparable performances for lowly recursive (or non-recurvise) XML data while the XB-tree outperforms the rest for highly recursive XML data.

3.6 Experiments - Part 2

This section presents the results of the experiments to evaluate the performance of the proposed path-based approach. We compare the path-based approach (by using TwigStack [19] and iTwigJoin [24]) with the state-of-the-art structural join solution, XB-tree based TwigStack [19], which shows the best performance in the previous section, and the latest path-based approach, BLAS [25] as well as the negation solution TwigStackList¬ [86]. All these approaches are implemented in C++. The operating system is Linux 2.4. The page size is set to 4 KB.

Datasets

Table 3.2 shows the characteristics of the experimental datasets which include the Shakespeare's Plays (SSPlays) [1], DBLP [4] and XMark benchmark [2]. Attributes are omitted for simplicity.

Datasets	Size	♯(Distinct Elements)	♯(Elements)
SSPlays	7.5 MB	21	179,690
DBLP	60.7 MB	32	1,534,453
XMark	61.4 MB	74	959,495

Table 3.2: Dataset Characteristics of Experiments 2

Query Workload

The query workload used in this section is shown in Table 3.3. The queries comprise short simple queries, long path queries, branch queries and queries involving negation relationships (Q1-Q12).

	Query	Dataset
Q1	//PLAY//TITLE	SSPlays
Q2	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR	SSPlays
Q3	//SCENE//STAGEDIR	XMark
Q4	//proceedings/booktitle	DBLP
Q5	//proceedings[/url]/year	DBLP
Q6	//people/person/profile[/age]/education	XMark
Q7	//closed_auction/annotation[//emph]//keyword	XMark
Q8	//regions/australia/item//keyword[//bold]//emph	XMark
Q9	//PLAY[NOT(/PROLOGUE)]/EPILOGUE//TITLE	SSPlays
Q10	//dblp/article[NOT(//url)]	DBLP
Q11	//person[NOT(/creditcard)]	XMark
Q12	//people/person[NOT(/age)]/profile/education	XMark

	Feature	♯ Nodes in Result
Q1	short simple path	1068
Q2	long simple path	2259
Q3	short simple path	6974
Q4	short simple path	3314
Q5	short branch query	5526
Q6	long branch query	7933
Q7	long branch query	13759
Q8	long branch query	74
Q9	negation	13
Q10	negation	14
Q11	negation	7618
Q12	negation	9568

Table 3.3: Query Workload

3.6.1 Storage Requirements

We first compare the space requirement of our proposed path-based solution with those of the XB-tree [19] and BLAS [25]. The original BLAS [25] approach is implemented in the relational database where each tuple in relations represents one element and the B^+-tree index is built on the attribute of p-label. In this chapter, our implementation of the BLAS groups the same p-labels. These p-labels are then indexed by using the B^+-tree, and each entry in the leaf node points to the set of XML nodes with this p-label. This slight modification does not affect query evaluation performance but can reduce the space requirement for BLAS [25]. Figure 3.14 illustrates the structure.

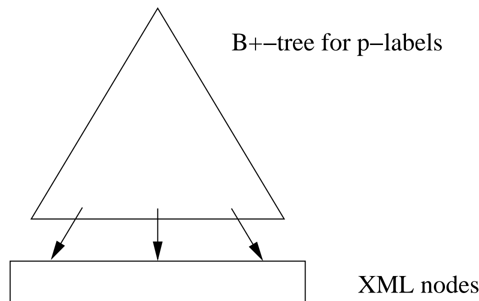


Figure 3.14: Implementation of BLAS

In both the XB-tree and the B^+-tree, we keep each node half full except for the root node. The page occupancy for the node lists in the path-based approach is also kept at 50%. In addition, to be consistent with the XB-tree and BLAS, the path-based solution also utilizes the interval-based node labeling scheme to assign node ids. The storage requirements are shown in Table 3.4.

It can be observed that BLAS [25] requires a little less space than our proposed approach does since it utilizes intervals to represent path information while ours uses bit sequences. For the path-based solution, the sizes of encoding tables are very small, and the space required by the path lists is determined by the degree

Datasets	XB-tree	BLAS	Path-based
SSPlays	8.0MB	6.5MB	6.56MB
DBLP	69.6MB	57.3MB	57.3MB
XMark	40.4MB	32.4M	33.2MB

BLAS		
P-Labels	D-Labels	Total
2.1KB	6.5MB	6.5MB
1.9KB	57.2MB	57.3MB
36.9KB	32.3MB	32.4MB

Path-based			
Encoding Tab	Path Lists	Node Lists	Total
0.24KB	5.9KB	6.5MB	6.56MB
0.38KB	9.1 KB	57.2MB	57.3MB
2.90KB	884.2KB	32.3MB	33.2MB

Table 3.4: Space Requirements

Datasets	♯(Distinct Path)	Path Id Size(Bytes)
SSPlays	40	5
DBLP	69	9
XMark	344	43

Table 3.5: Storage for Path Ids

of regularity of the structures of the XML documents (see Table 3.5). Real-world datasets typically have a regular structure, and thus have fewer distinct paths (40 distinct paths in SSPlays and 69 in DBLP) compared to the 344 distinct paths in the synthetic XMark dataset. Since the number of bits in the path id is given by the number of distinct paths, the path ids for SSPlays and DBLP are only 5 and 9 bytes respectively. In contrast, the irregular structure in XMark needs 43 bytes for the path id.

We also observe that the size of the path lists is relatively small compared with that of the node lists. Even for the most irregular structure dataset XMark, path lists size takes only 2.7% of node lists size (884K and 32M respectively). This feature fundamentally guarantees the low cost of path join.

Next, we investigate the query evaluation performance of the path-based approach and compare it with the XB-tree based holistic join [19], BLAS [25] and TwigStackList¬ [86].

3.6.2 Structural Join

In this set of experiments, we demonstrate the effectiveness of the path-based solution on evaluating structural join queries.

Effectiveness of Path Join

This part examines the effectiveness of the path join algorithm in filtering out elements that are not relevant for subsequent node join.

A metric called "Filtering Efficiency" is first defined to measure the filtering ability of path join. This metric gives the ratio of the number of nodes after path join over the total number of nodes involved in the query. That is, given a query Q, we have:

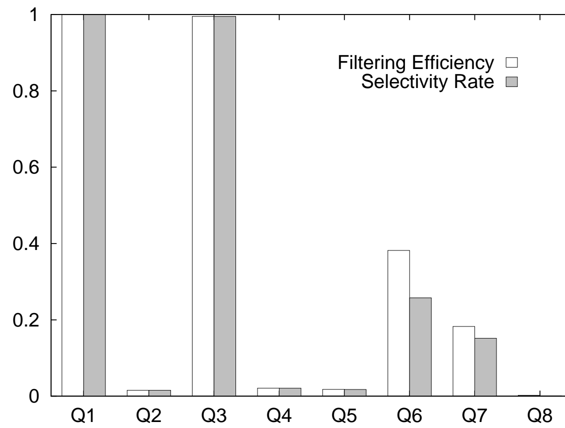
$$Filtering\ Efficiency = \frac{\sum |N_i^p|}{\sum |N_i|}$$

where $|N_i^p|$ denotes the number of instances for node N_i after path join and $|N_i|$ denotes the total number of instances for N_i in the projected tree of the query.

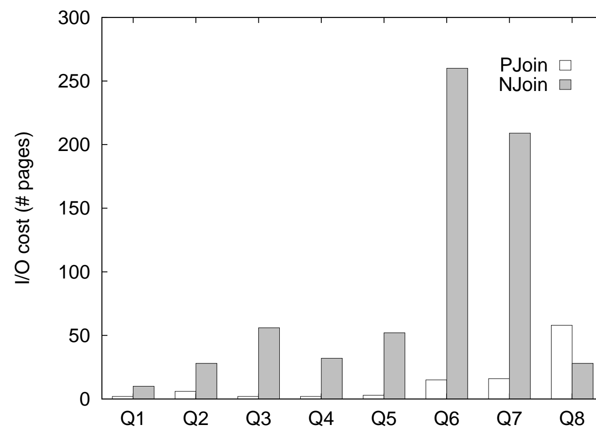
We also define "Selectivity Rate" to reflect the percentage of nodes in the result set compared to the original number of nodes involved in the query. Given a query Q, we have:

$$Selectivity\ Rate = \frac{\sum |N_i^n|}{\sum |N_i|}$$

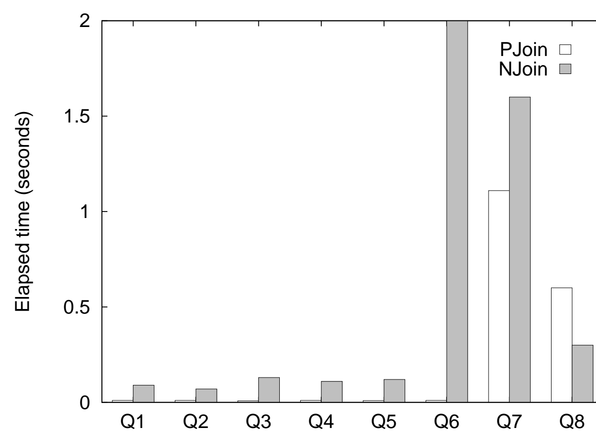
where $|N_i^n|$ denotes the number of instances for node N_i in the result set after a node join and $|N_i|$ is the total number of instances for N_i in the projected tree of the query.



(a) Filtering Efficiency vs. Selectivity Rate



(b) PJoin and NJoin (I/O cost)



(c) PJoin and NJoin (Time)

Figure 3.15: Effectiveness of Path Join

The effectiveness of path join can be measured by comparing values of Filtering Efficiency and Query Selectivity of the queries. Based on the definitions of these two metrics, we can see the closer the two values are, the more effective the path join is for the query. The optimal case is achieved when Filtering Efficiency is equivalent to Query Selectivity, indicating that path join has effectively filtered out all irrelevant elements for the subsequent node join.

Figures 3.15(a) compares Filtering Efficiency with Query Selectivity for queries Q1 to Q8. Except for queries Q6, Q7 and Q8, the queries have the same values for the two metrics. This shows that path join has effectively removed all irrelevant elements for the node join.

Queries Q6, Q7 and Q8 have higher Filtering Efficiency values compared to Query Selectivity. This indicates that the path join algorithm does not produce exact *Pid* sets for the subsequent node join for these queries. As we analyzed in Section 3.3.3, the *Pid* sets associated with nodes after path join may not be the exact *Pid* sets for branch queries. Since Q6, Q7 and Q8 are all branch queries, this result is expected. Note that path join still remains efficient in eliminating unnecessary path types even for branch queries, which can be seen from the close values of Filtering Efficiency and Query Selectivity of queries Q5, Q6, Q7 and Q8 (all are branch queries, and the two values are the same for Q5).

Figures 3.15(b) and (c) compare the I/O cost and elapsed time of path join and node join respectively. Both graphs show that the cost of path join is very marginal for the majority of the queries compared to that of node join. This is because the size of the path lists involved in the query is much smaller than the size of the node lists (recall Table 3.4).

In Figures 3.15(b) and (c), the costs of path join for queries Q1 to Q5 are negligible because of the regular structures of SSPlays and DBLP. Path join is more

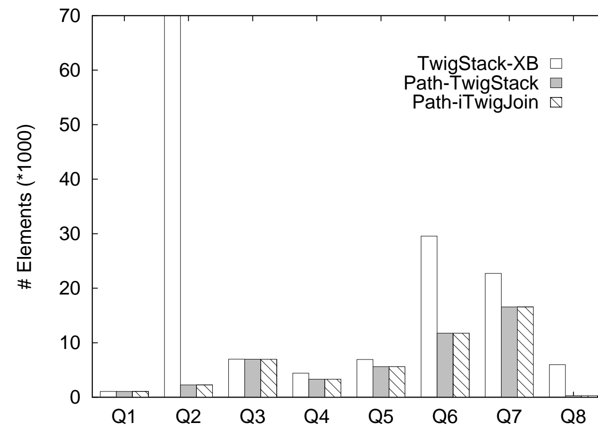
expensive for the queries over the XMark dataset (Q6 to Q8) due to the irregular structure of the dataset, which results in a larger number of path types and longer path ids. Among these queries on the synthetic dataset (Q6 to Q8), query Q8 is the only one where the cost of path join is greater than the node join. This can be explained by the low selectivity of Q8 (74 nodes in result, Table 3.3), which directly contributes to the low cost of node join. Finally, the result in Figure 3.15(a) clearly demonstrates that path join remains effective in filtering out a large number of elements for queries even with the influence of irregularity in synthetic dataset.

XB-Tree vs. Path-Based Approach

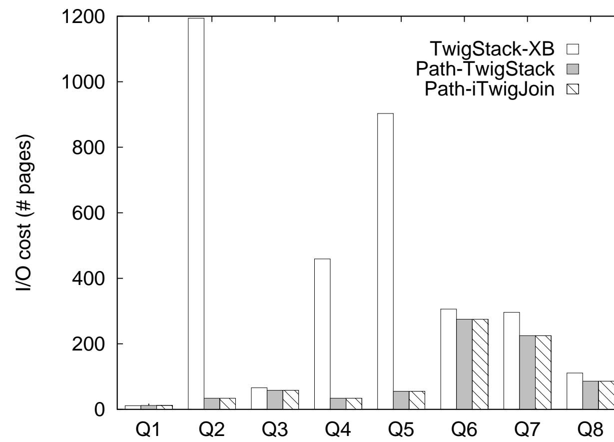
In this set of experiments, we compare the performance of the path-based approach with the XB-tree based holistic join solution TwigStack [19]. For path based solution, we use both TwigStack [19] and iTwigJoin [24] to carry out the node join. Given an XML query, the XB-tree based structural join solution TwigStack calculates the matching instances of all nodes involved in the query pattern. As a result, we assume that for all queries all nodes involved in the query pattern are projected nodes.

The metrics used here to measure performance are total number of elements accessed, I/O cost (the number of pages), and total elapsed time. Figure 3.16 shows that both path-based approaches (Path-TwigStack and Path-iTwigJoin) perform significantly better than XB-tree based holistic join. This is because path join is able to greatly reduce the actual number of elements retrieved as shown in Figure 3.15(a).

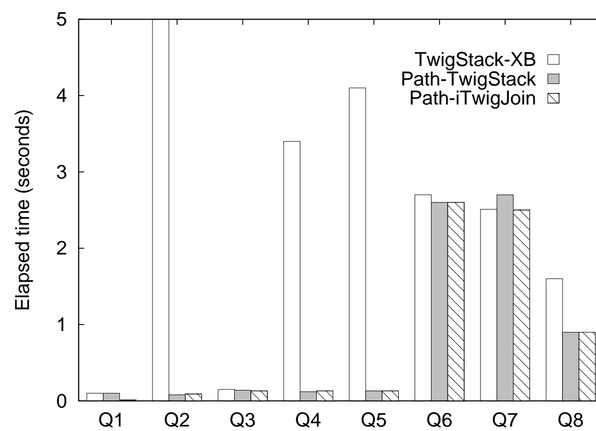
We observe that the Path-TwigStack and Path-iTwigJoin have almost identical query evaluation performances in terms of number of elements accessed, I/O costs and elapsed time. This is because the pre-processing step, path join, eliminates



(a) Elements Accessed



(b) I/O Cost



(c) Elapsed Time

Figure 3.16: XB-tree Based Holistic Join vs. Path Based Structural Join

most irrelevant elements. As a result, although the path and level information are contained in the partitioned streams involved in the node join, the Path-iTwigJoin cannot reach better performance since almost all elements remained in the streams contribute to the final results. For this reason, we only use Path-TwigStack method in the later experiments.

In addition, we note that the underlying data storage structure of the path-based approach affects the query performance. For queries Q4 and Q5, the I/O cost is smaller than the number of elements accessed in the path-based approaches (see Figure3.16(a) and (b)). This is because the path-based approaches cluster node records according to their paths. This further reduces I/O cost during data retrieval. In contrast, I/O cost for the XB-tree is determined by the storage distribution of matching data. In the worst case, the elements to be accessed are scattered over the entire list, thus leading to high I/O cost.

Effect of Parent-Child Relationships

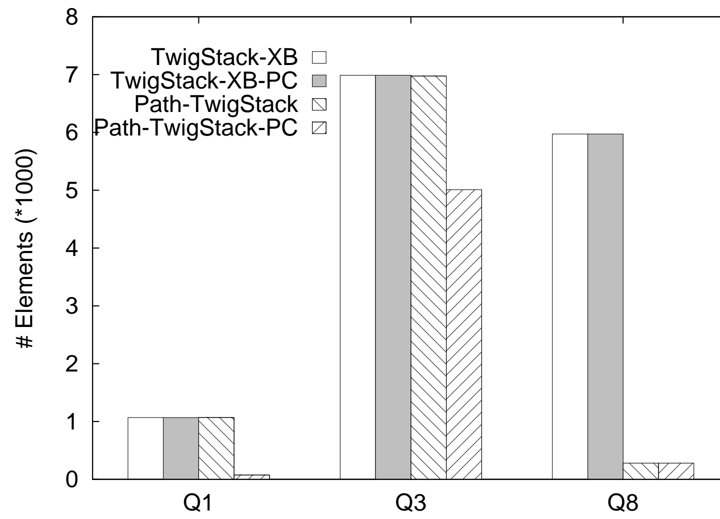
To examine the effect of parent-child relationships, we replace some ancestor-descendant edges in queries Q1, Q3 and Q8 with parent-child relationships (see Table 3.6). Figure 3.17 shows the result.

	Query	Dataset
Q1pc	//PLAY/TITLE	SSPlays
Q3pc	//SCENE/STAGEDIR	SSPlays
Q8pc	//regions/australia/item//keyword[/bold]/emph	XMark

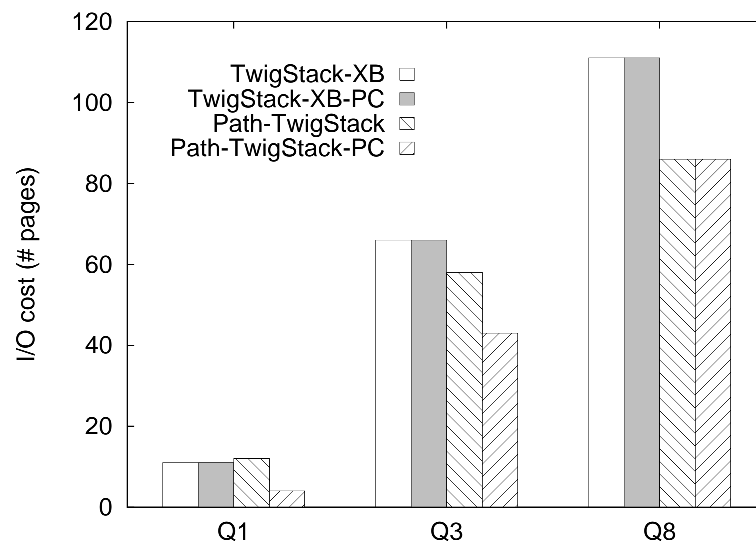
	Feature	♯ Nodes in Results
Q1pc	parent-child	74
Q3pc	parent-child	5010
Q8pc	parent-child	74

Table 3.6: Parent-Child Queries

The XB-tree based holistic join utilizes the same method to evaluate parent-



(a) Elements Accessed



(b) I/O Cost

Figure 3.17: Parent-Child Queries

child queries and ancestor-descendant queries. Therefore, XB-tree based holistic join has the same evaluation performance for parent-child and ancestor-descendant queries. To avoid incorrect result, each parent-child edge is (inexpensively) verified before it is output.

In contrast, the proposed path-based approach can check for parent-child edges during path join. This task is achieved by looking up the encoding table (see Figure 3.1(b)). In the case where the results of parent-child queries are subsets of the ancestor-descendant counterparts, the cost to evaluate queries may be further reduced since fewer elements are involved in the node join. For example, queries Q1pc and Q3pc have smaller result sets compared to Q1 and Q3 (see Table 3.3) respectively. Thus Q1pc and Q3pc show better performance in Figure 3.17.

Effect of Value Predicates

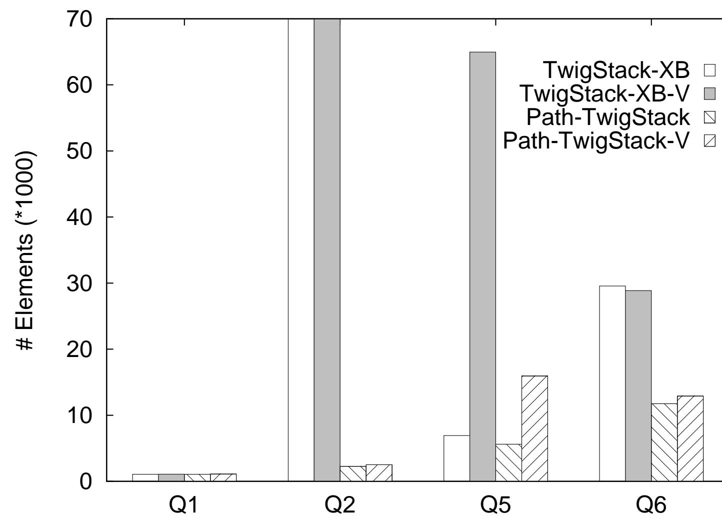
Finally, we investigate how the proposed approach and the XB-tree perform for queries involving value predicates. We add value constraints on queries Q1, Q2, Q5 and Q6 respectively (see Table 3.7). The results are shown in Figure 3.18.

	Query	Dataset
Q1v	//PLAY//TITLE="ACT II"	SSPlays
Q2v	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR="Aside"	SSPlays
Q5v	//proceedings[/url]//year="1995"	DBLP
Q6v	//people/person[/age="18"]/profile/education	XMark

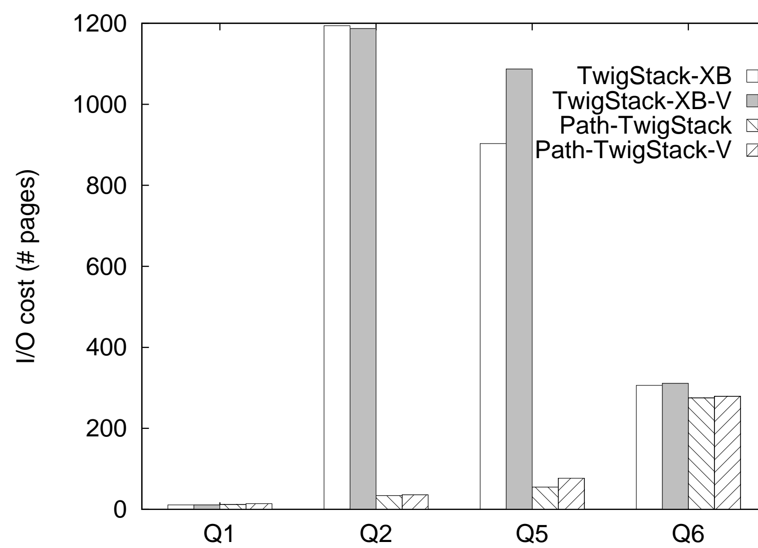
	Feature	♯ Nodes in Result
Q1v	value predicate	111
Q2v	value predicate	1044
Q5v	value predicate	432
Q6v	value predicate	2336

Table 3.7: Value Predicates Queries

When evaluating XML queries involving value predicates, the path-based solution first carries out a path join to process the structural aspects of the queries.



(a) Elements Accessed



(b) I/O Cost

Figure 3.18: Queries with Value Predicates

To determine the final set of results, the subsequent node join retrieves the value nodes and the element nodes obtained by the path join. Therefore, the path-based solution needs to access more nodes to evaluate the value predicates in the queries compared to the corresponding queries without value predicates. This can be observed in Figure 3.18.

The XB-tree based holistic join solution treats value nodes the same way as element nodes. The additional value predicates incur more costs during the retrieval of nodes. However, the value constraints may reduce the total number of element nodes accessed. This is because the XB-tree approach employs the XB-tree index to search for matching nodes. Thus, it may skip some element nodes that match the structural query pattern but not the value predicates. Figure 3.18 shows that the addition of value predicates have different effects on performance for Q5 and Q6.

Overall, the evaluation of structural patterns still dominates the query performance even for queries involving value predicates. This is shown clearly in Figure 3.18.

In summary, the comparative experiments between XB-tree based holistic join and our proposed approach demonstrate that the path-based solution outperforms existing structural join methods for the following reasons:

1. Path join efficiently filters out nodes with path types that are not relevant to the subsequent node join.

2. The cost of path join is marginal compared to node join in the majority of queries.

3. Element records are clustered according to path types, which further reduces I/O cost during element retrieval.

BLAS vs. Path-Based Approach

In this part, we first give an overview of the BLAS approach [25], and then compare our proposed solution with BLAS [25].

BLAS Overview. BLAS [25] proposes a p-labeling system to process XML queries, especially for suffix queries. Suffix queries start with an optional descendant axis followed by a set of child axes. For example, path queries $//A/B/C$ and $/A/C$ are both suffix queries. From the definition, it can be seen that the suffix query is a subset of the simple XML query.

BLAS [25] assigns each suffix path (whether it occurs in the XML data or not) a p-label of the interval format $(p1, p2)$. For any two suffix paths P and Q, Q is a suffix of P if and only if the p-label of Q contains that of P, that is, $Q.p1 < P.p1 < P.p2 < Q.p2$.

In the p-labeling system, each XML node is associated with the p-label of the root-to-node path where the node occurs, and the B^+-tree is used to index all the p-labels (see Figure 3.14). Given a suffix query P, BLAS [25] first calculates its p-label interval, then searches the B^+-tree to find all paths that have P as a suffix. Finally, the nodes associated with the result p-labels (root-to-node paths) are retrieved to answer query P.

Example 3.22: Given a suffix query $P = //X/Y$ with p-label (50,100), the range search is issued on the B^+-tree to find the p-labels which are contained in this interval. These result p-labels must represent all the paths with P as a suffix according to the p-label containment relationship. For example, we may get paths $/A/X/Y(60, 79)$ and $/B/X/Y(91, 99)$. Finally, the set of Y nodes with p-labels $(60, 79)$ and $(91, 99)$ are retrieved as the results of the given suffix query P. □

When a branch query or a simple query containing descendant axes in between nodes is issued, BLAS [25] first decomposes the given query into a set of suffix queries, then join the intermediate results of these suffix queries. In [25], three decomposition methods, namely Split, Push-up and Unfold algorithms are designed, and Push and Unfold algorithms show better performance than Split method. Since Unfold relies on XML DTD or Scheme information to optimize queries while Push-up does not, we adopt the Push-up decomposition method in our comparative experiments. Figure 3.19 gives an example of Push-up decomposition method and details can be found in [25].

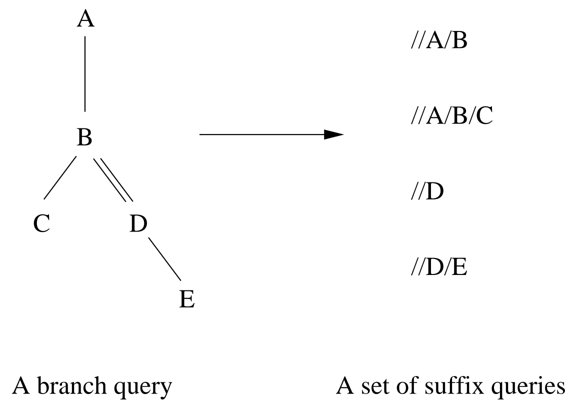


Figure 3.19: Decomposing a Branch Query into a Set of Suffix Queries

Comparative Experiments. The assumptions behind BLAS [25] and TwigStack [19] are different, and hence these two methods are not directly comparable. Given an XML query, TwigStack [19] calculates the matching instances of all nodes involved in the query pattern, that is, all the nodes are projected nodes. On the other hand, BLAS [25] considers only the last node reached via axes in the query as the projected node. Thus it avoids carrying out join between nodes for suffix queries.

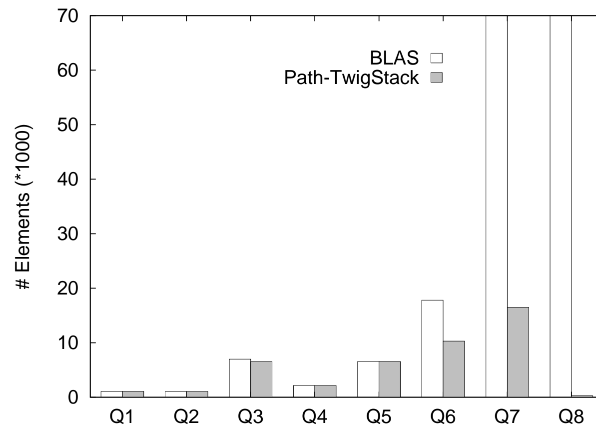
In this part, we compare our proposed solution with BLAS [25] using all the

structural join queries in Table 3.3 (Q1 to Q8) and assume only the last node in each query is the projected node. Figure 3.20 shows the comparison results.

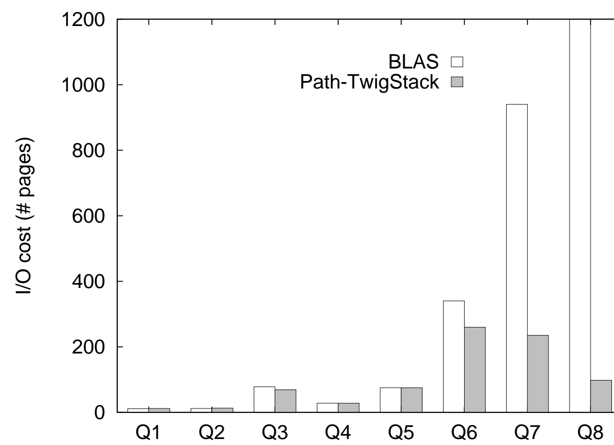
We observe that the path-based solution and BLAS [25] have the almost same performance results for suffix queries (Q2, Q4) since both of them can directly locate the matching elements with the help of path information. But outside of suffix queries, path-based solution outperforms BLAS [25] in the following two aspects:

For simple queries containing descendant axes in between nodes (Q1, Q3), the path join in the path-based solution can produce the exact path id sets for all nodes in the query pattern. Since only the last node is the projected node, instead of carrying out node join, we can directly retrieve result nodes based on the path ids obtained from path join. In contrast, BLAS [25] must split the descendant axes and perform the join, which affects query evaluation performance. For example, the path-based solution can directly retrieve results on query Q3 while BLAS must carry out the join between the "*SCENE*" and "*STAGEDIR*" elements.

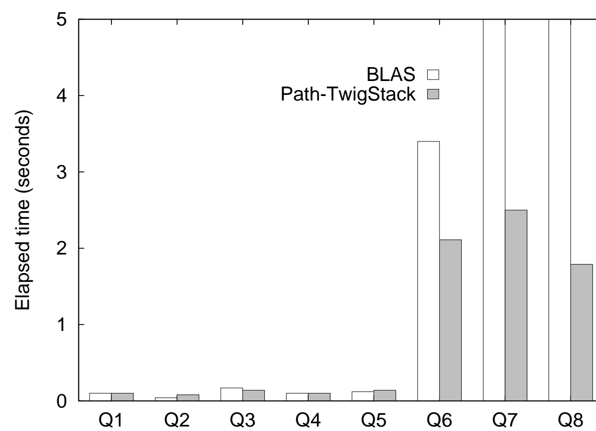
For branch queries (Q5-Q8), both approaches must perform join operations. However, the path join in the path-based solution may generate the smaller candidate element sets and have a smaller number of joins. For instance, BLAS [25] decomposes query Q8 into four suffix queries: *//region/australia/item*, *//keyword*, *//bold* and *//emph*, and utilizes p-labels to retrieve the results for the first suffix query. In contrast, path join associates exact pids with the nodes *region*, *australia*, *item* and *keyword* which occur on the trunk part of Q8. In addition, the path id sets associated with the elements *bold* and *emph* also satisfy the root-to-leaf-paths *//regions/australia/item//keyword//bold* and *//regions/australia /item//keyword//emph* respectively due to the method of path join, leading to the smaller node sets involved in the later node join compared with the full sets used



(a) Elements Accessed



(b) I/O Cost



(c) Elapsed Time

Figure 3.20: BLAS vs. Path-Based Solution

in BLAS [25]. The efficiency of path join on filtering out unnecessary elements for Q8 can be found in Figure 3.15(a).

Note that the performance results of the path-based solution in Figure 3.20 are better than its counterparts in Figure 3.16. This is because we avoid performing node join on simple queries. For branch queries, node join can be carried out on part of the elements in the query pattern since only one element is the projected node. The elements involved in node join can be chosen from the trunk path and all branch paths (one path one selected node). This feature also results in a smaller number of joins compared with BLAS [25]. For example, in query Q8, only the node sets of *keyword*, *bold* and *emph* are needed to take part in the node join after path join.

In summary, our proposed solution outperforms the BLAS [25] approach since the path-based labeling scheme captures more efficiently the path information where XML elements occur than p-labels do. Our proposed path join can produce the exact results for simple queries without join while BLAS can only achieve this for suffix queries. For branch queries, path join can generate smaller element sets and fewer joins compared with BLAS [25].

3.6.3 Negation

In this part, we explore the evaluation performance of proposed path-based solution on negation queries.

Effectiveness of Path Join+

Similar to the structural join part, we first check the effectiveness of path join+. We still employ the metrics "Filtering Efficiency" and "Selectivity Rate" to measure the ability of $PJoin+$ to filter out unnecessary elements. The definitions of two

metrics are identical to those in structural join part except that we use "PJoin+" and "NJoin+" to replace "PJoin" and "NJoin" respectively. The result is shown in Figure 3.21.

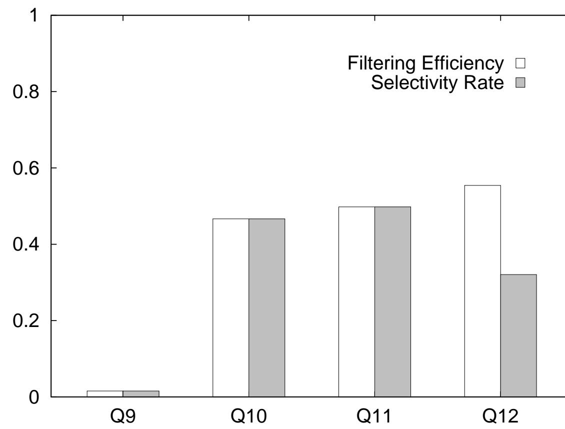
As expected, the PJoin+ remains as effective as PJoin dose in structural part. For queries Q9, Q10 and Q11, "Filtering Efficiency" and "Selectivity Rate" have the same values (see Figure 3.21(a)). This indicates that path join+ has deleted all unnecessary elements. In other words, the node join+ followed by will not access any element that does not contribute to the final results, leading to the optimal query evaluation performance. Moreover, the I/O cost and elapsed time of path join+ (see Figure 3.21(b) and (c)) are marginal compared with node join+ for most queries. This result is consistent with that in structural join part.

Comparative Experiments.

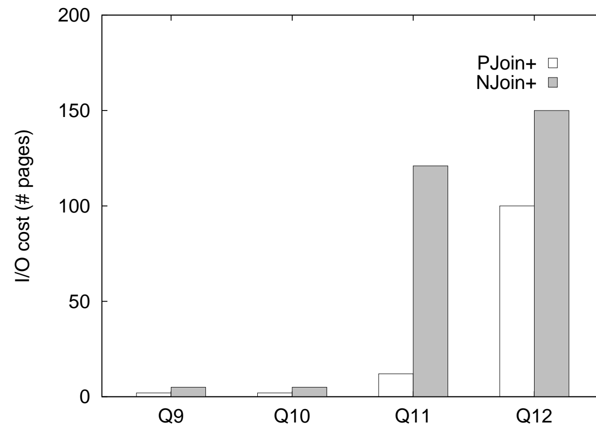
In this part, we compare our proposed solution with TwigStackList¬ [86] by using all negation queries in Table 3.3 (Q9 to Q12). The node join+ algorithm in our proposed path-based approach is identical to TwigStackList¬. That is, we use path join+ to minimize the element candidate sets, then apply the holistic negation join solution on evaluating queries. Figure 3.22 shows the comparison results.

We observe that the path-based solution outperforms TwigStackList¬ [86] by using the path join+ algorithm. This is because of the similar reason shown in structural join part. TwigStackList¬ [86] is specially designed to reduce the intermediate result sizes, and it may access all the elements of the streams involved in the queries. Our solution, in contrast, only needs to retrieve the elements that pass the path join+ examination.

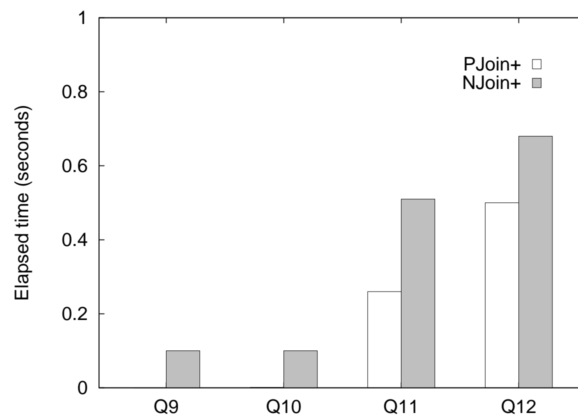
For example, path join+ filters out about 95% elements for query Q9 as shown in Figure 3.21(a). As a result, the node join+ method followed by retrieves much



(a) Filtering Efficiency vs. Selectivity Rate

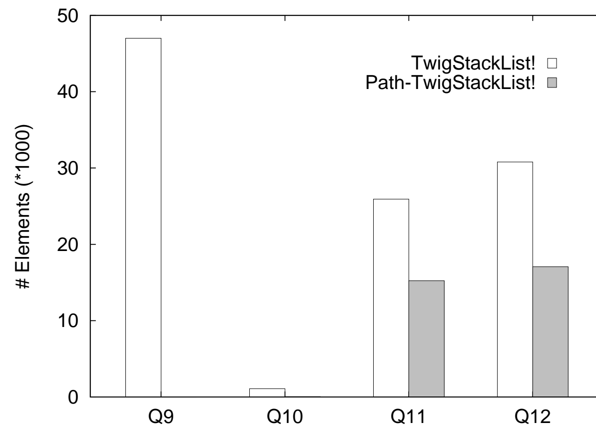


(b) PJoin+ and NJoin+ (I/O cost)

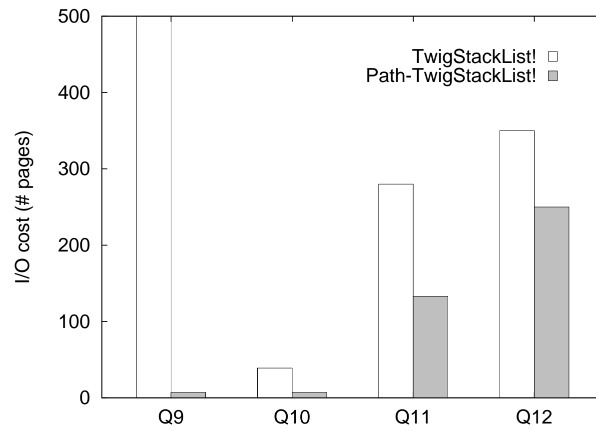


(c) PJoin+ and NJoin+ (Time)

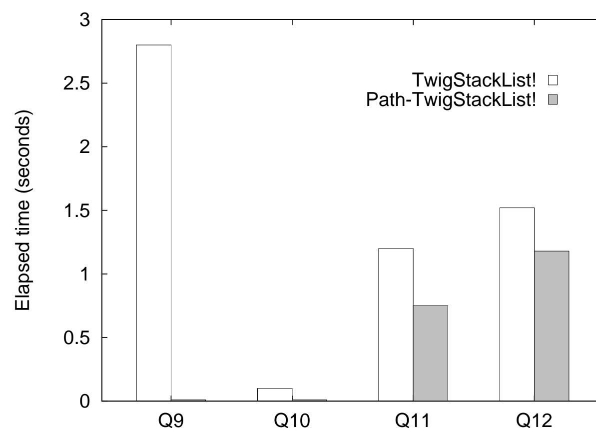
Figure 3.21: Effectiveness of Path Join+



(a) Elements Accessed



(b) I/O Cost



(c) Elapsed Time

Figure 3.22: TwigStackList¬ vs. Path-Based Negation Join

less number of elements than TwigStackList¬ does in Figure 3.22(a) which must read in the full sets of elements when evaluating the query.

3.7 Conclusion

In this chapter, we have presented a new paradigm for processing structural join and negation in XML queries. The proposed solution includes a path-based labeling scheme which associates each element node with path information and a path join (path join+) algorithm that is able to compute the minimal sets of elements required for the subsequent node join (node join+). Experimental results clearly show that the proposed approach outperforms the existing holistic structural join method TwigStack [19], the path index approach BLAS [25] as well as negation solution TwigStackList¬ [86].

CHAPTER 4

A Statistical Query Selectivity Estimator for XML Data

4.1 Introduction

The optimization of XML queries requires an accurate and compact structure to capture the characteristics of the underlying data. In this chapter, we develop a comprehensive solution to estimate the query selectivity of general XML query patterns, which are the combinations of any linear and twig queries. We extract highly summarized information, namely, *Node Ratio* (NR) and *Node Factor* (NF) from every distinct parent-child path. When evaluating an XML query, statistical information is recursively aggregated by using the path-independence assumption to estimate the frequency of the target node. Compared with the existing solutions, our method utilizes statistical data that is compact and yet proves to be sufficient in estimating the selectivity of queries for regularly distributed XML data.

We also propose a method to augment the statistical model with histograms to reduce estimation errors when the XML data is skewed. We construct histograms for selected parent-child paths to capture the characteristics of the underlying data

distribution based on the interval-based numbering scheme that is widely used for structural join. Experimental results demonstrate the effectiveness of histograms.

The rest of this chapter is organized as follows. Section 4.2 introduces the background knowledge. Section 4.3 describes the proposed estimation method. The histogram-based solution is introduced in Section 4.4. Section 4.5 gives the experimental results. We conclude in Section 4.6.

4.2 Preliminary

4.2.1 Problem Definition

In this chapter, we focus on non-recursive tree structured XML data and XML queries involving only element structure, that is, queries without value predicates. In addition, to clarify the projected node in the given XML query, the query pattern must specify the target element, which is the node whose size we want to estimate. This is because different nodes may have different cardinalities in the result sets. For example, the result of the query "$//department/professor$" may contain one department and ten professors.

The selectivity estimation problem can be stated formally as follows:

Given a non-recursive XML tree $X = (V_X, E_X)$ where V_X and E_X denote the sets of nodes and edges between nodes respectively, and an XML query $Q = (V_Q, E_Q)$ with target node T, $V_Q \subset V_X$, $T \in V_Q$, we estimate the number (selectivity) of elements with tag T satisfying Q in X.

Example 4.1: Given an XML linear query shown in Figure 4.1 (the first query) and the XML instance in Figure 4.3 with target node C, the selectivity of element C is two since there are two C occurrences matching the query pattern in the XML data. □

4.2.2 Taxonomy

This section provides the taxonomy of XML queries. In the chapter, the XML query language, either XPath [10] or XQuery [7], could be used since both of them have the ability to describe the XML element structure patterns.

Simple Path (Linear Path). Given an XML query Q, if all nodes in Q are connected sequentially, then Q is called a simple (or linear) path. The relationship between any two adjacent nodes can be an ancestor-descendant or a parent-child relationship.

Basic Path. Given a simple path query Q, if Q only contains two nodes, then Q is called a basic path.

Twig (Branch) Query. If the root node of a given XML query Q has more than one immediate child nodes, then Q is called a twig (branch) query, or simply a twig.

Simple Twig. Given a twig query Q, if every branch of Q is a simple path, then Q is known as a simple twig.

General Path. A general path is all the possible combinations of *simple paths* and *twigs*.

Example 4.2: Figure 4.1 illustrates the different queries. The single line and double line denote the parent-child and ancestor-descendant relationships respectively. □

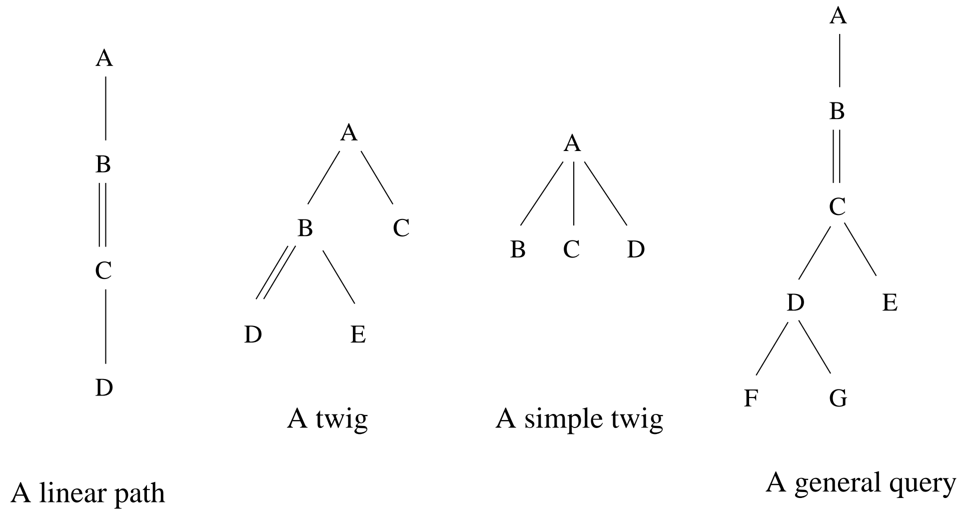


Figure 4.1: Classification of XML Queries

4.3 Estimation Method

This section describes the building blocks of the proposed query selectivity estimator. In this proposed estimation system, we collect the statistical information of each basic parent-child path. Given an XML query Q to be estimated, we first decompose Q into a set of basic paths, then recursively aggregate the statistical information to estimate the query selectivity. In the rest of this section, we first introduce the query decomposition and statistics collection, and then present statistics aggregation methods and the estimation algorithm.

4.3.1 Query Decomposition

Given a general XML query Q, we can decompose it into a linear path and a twig which represent the "trunk" and "branch" parts of Q respectively. Similarly, the branch part can be further decomposed into a set of queries Q_i, each of which is a branch of Q. The process is then repeated for each Q_i if Q_i is not a simple path. Finally, the resulting set of simple path queries can be transformed into a set of basic paths.

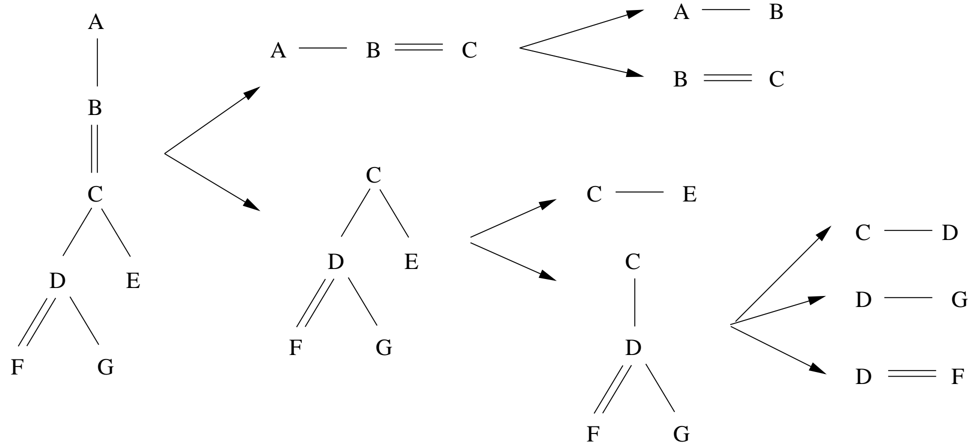


Figure 4.2: Decomposing a General Query into a Set of Basic Queries

Example 4.3: Figure 4.2 demonstrates how a general path query can be systematically decomposed into a set of basic path queries. □

4.3.2 Summary Statistics

Given a path query Q in an XML dataset X, we denote the root node in the path as R and the target node in Q as T. The target node T refers to the node whose selectivity we want to estimate. We use $f(node)$ to denote the frequency of *node* in X and $f(node|path)$ represents the frequency of *node* occurring on the *path* in the dataset X.

Example 4.4: In Figure 4.3, we have $f(C) = 4$ and $f(C|A/B/C/D)$=2. □

In this estimation system, two important summary variables, namely $Node\ Ratio$ (NR) and $Node\ Factor$ (NF), are captured. The variable NR indicates the ratio of the number of occurrences of a root node R in some path P to the total occurrences of R in an XML dataset while the variable NF gives the average number of node T for a given root node R in a path P.

Definition 4.1 (Node Ratio - NR) *Let X be an XML dataset and P a general*

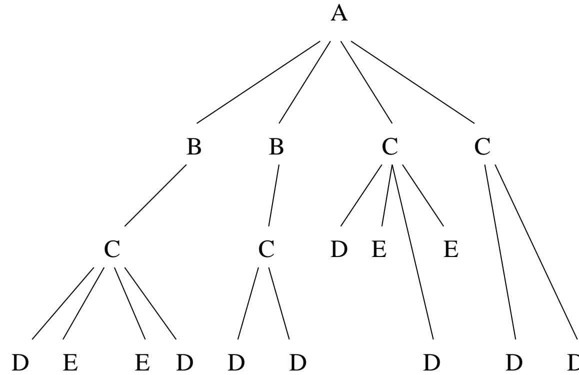


Figure 4.3: An XML Instance

path in X, and R the root node in P. We define the ratio of the frequency of R in P to the frequency of R in X as the node ratio NR of R in P, which is given by:

$$NR(R|P) = f(R|P)/f(R)$$

Example 4.5: In Figure 4.3, $NR(B|B/C/D) = f(B|B/C/D)/f(B) = 2/2 = 1$ and $NR(C|C/E) = f(C|C/E)/f(C) = 2/4 = 0.5$. □

Definition 4.2 (Node Factor - NF) *Let X be an XML dataset and P a general path in X, and R and T the root node and target node in P respectively. We define the ratio of the frequency of T in P to the frequency of R in P as the node factor NF of T in P. That is:*

$$NF(T|P) = f(T|P)/f(R|P)$$

Example 4.6: Consider Figure 4.3 again. We have $NR(C|B/C/D) = f(C|B/C/D)$ $/f(B|B/C/D) = 1$ and $NR(B|A/B) = f(B|A/B)/f(A|A/B) = 2$. □

From the above definitions, it is easy to obtain:

$$f(R|P) = f(R) * NR(R|P)$$
$$f(T|P) = f(R) * NR(R|P) * NF(T|P)$$

The above equations can be utilized to estimate query selectivity. To distinguish estimated results from actual values, we use $S_P(R)$ and $S_P(T)$ to represent the estimated selectivity of node R and T occurring in the path query P, and we have:

$$\begin{aligned} S_P(R) &= f(R) * NR(R|P); \\ S_P(T) &= f(R) * NR(R|P) * NF(T|P) \end{aligned} \tag{4.1}$$

In this estimation system, we capture the frequency of every distinct element tag, the NR and NF for each distinct basic parent-child path. This information is then utilized to estimate the selectivity of XML queries.

Node	Frequency
A	1
B	2
C	4
D	8
E	4

Parent-Child Paths	NR	NF
A/B	1	2
A/C	1	2
B/C	1	1
C/D	1	2
C/E	0.5	2

Figure 4.4: NR and NF Values for Parent-Child Paths

Example 4.7: Figure 4.4 shows the statistical information collected on the XML instance of Figure 4.3. For the parent-child paths, the root node R and the target node T are the parent node and the child node in the path respectively. Note that from the definitions, we always have $NR \leq 1$ and $NF \geq 1$. □

The statistical information in our proposed solution can be collected by simply carrying out a depth-first tree traversal on the XML data. During this procedure, the frequencies of each distinct element tag and parent-child path are counted.

After tree navigation, we then compute all the corresponding NR and NF values. Since XML data is organized in the tree depth-first format, a sequential scan is enough to obtain all necessary statistics.

4.3.3 Statistics Aggregation Methods

Next, we explain how the NR and NF information of basic parent-child paths can be aggregated to estimate the size of the various path queries. The estimation methods are based on the following assumption:

- **Basic Path Independence Assumption:** The distribution of a basic path p in an XML document X is independent of the distribution of other basic paths. In other words, the distribution of the child node of a basic path only depends on its parent node.

Example 4.8: In Figure 4.3, the distribution of basic path C/D is independent of its incoming paths, B/C and A/C, and the sibling path C/E. That is, each C element contains two child nodes D regardless that the C has parent node B or A, or child node E. □

The idea behind the proposed estimation approach is that we recursively aggregate the NR and NF values of basic parent-child paths and then utilize formula 4.1 to estimate the selectivity. There are two aggregation methods: serialization and parallelization. We introduce the methods below.

Serializing Simple Paths

Suppose we have two simple paths p_1 and p_2, and their corresponding $NR(R_1|p_1)$, $NF(T_1|p_1)$, $NR(R_2|p_2)$ and $NF(T_2|p_2)$, where R_i and T_i are the first and last nodes in path p_i. If the target node T_1 has the same label as that of the node R_2, we

can connect the two paths p_1 and p_2 sequentially and eliminate the redundant node R_2. The result is also a simple path, denoted as $p_1 \cdot p_2$. For instance, if $p_1 = A/B, p_2 = B/C$, then $p_1 \cdot p_2 = A/B/C$.

To calculate $NR(R_1|p_1 \cdot p_2)$ and $NF(T_2|p_1 \cdot p_2)$, we first compute $NF(T_1|p_1 \cdot p_2)$. From the definition of NR and NF, it is easy to see that the accurate value of $NF(T_1|p_1 \cdot p_2)$ should be $NF(T_1|p_1)$ * $(f(T_1|p_1 \cdot p_2)\ /f(T_1|p_1))$. Taking into consideration the Basic Path Independence Assumption, we use $f(T_1|p_2)/f(T_1)$, which is $NR(R_2|p_2)$, to replace $f(T_1|p_1 \cdot p_2)/f(T_1|p_1)$. Hence, the value of $NF(T_1|p_1 \cdot p_2)$ is estimated as $NF(T_1|p_1)$*$NR(R_2|p_2)$.

Next, we use two examples in Figure 4.3 to explain how to obtain $NR(R_1|p_1 \cdot p_2)$ and $NF(T_2|p_1 \cdot p_2)$.

Example 4.9: Case 1. Estimating $S_Q(D)$ where $Q = B/C/D$.

As introduced above, we have $NF(C|Q) = NF(C|B/C) * NR(C|C/D) = 1$. Here, the purpose of $NR(C|C/D)$ is to filter out those C elements which are under B but do not contain D. If the result $NF(C|Q)$ is greater than or equal to 1 (as shown in this case), then $f(C|Q)$ is greater than or equal to $f(B|Q)$ (by NF definition). Assuming that all C elements are uniformly distributed under all B elements, we can deduce that $f(B|Q)$ should remain unchanged as $f(B|B/C)$. As a result, we set $NR(B|Q) = NR(B|B/C)$. Similarly, we can get $NF(D|Q) = NF(C|Q) * NF(D|C/D)$. Since the values (after calculation) of $NR(B|Q)$ and $NF(D|Q)$ are 1 and 2 respectively, we finally get $S_Q(B) = f(B) * NR(B|Q) = 2$ and $S_Q(D) = f(B) * NR(B|Q) * NF(D|Q) = 4$. □

Example 4.10: Case 2. Estimating $S_Q(E)$ where $Q = B/C/E$.

We have $NF(C|Q) = NF(C|B/C) * NR(C|C/E) = 0.5$. In this case, $NF(C|Q)$ is less than 1, which violates the definition of NF (the number of the target node must be greater than or equal to the number of the root node in the same path).

This shows the addition of path C/E actually filters out not only those C elements under B, but also B elements as shown in Figure 4.3. To handle this situation, we set $NR(B|Q) = NR(B|B/C) * NF(C|Q)$ since $NF(C|Q)$ reduces the number of B elements. After that, the value of $NF(C|Q)$ is set to 1. This is because we assume that the remaining C elements (which are not filtered out by the addition of path C/E) are uniformly distributed under B. Thus, the number of B elements should be reserved as many as possible and $NF(C|Q)$ must be 1. Similarly, $NF(E|Q) = NF(C|Q) * NF(E|C/E) = NF(E|C/E)$. Finally, we estimate $S_Q(B) = f(B) * NR(B|Q) = 1$ and $S_Q(E) = f(B) * NR(B|Q) * NF(E|Q) = 2$. □

From the two examples above, we can find that given two simple paths p_1, p_2 and their corresponding $NR(R_1|p_1)$, $NF(T_1|p_1)$, $NR(R_2|p_2)$ and $NF(T_2|p_2)$, if the value of $NR(T_1|p_1 \cdot p_2)$ is greater than or equal to 1, then some elements T_1 occurring in p_1 are filtered out with the addition of the path p_2. Otherwise, the number of R_1 is reduced. As a result, we summarize the following formula:

Given two paths p_1 and p_2 where $T_1 = R_2$

$$\begin{aligned}
& NF(T_1|p_1 \cdot p_2) = NF(T_1|p_1) * NR(T_1|p_2) \\
& if \quad (NF(T_1|p_1 \cdot p_2) \geq 1) \\
& \quad NR(R_1|p_1 \cdot p_2) = NR(R_1|p_1) \\
& \quad NF(T_2|p_1 \cdot p_2) = NF(T_1|p_1 \cdot p_2) * NF(T_2|p_2) \\
& else \\
& \quad NR(R_1|p_1 \cdot p_2) = NR(R_1|p_1) * NF(T_1|p_1 \cdot p_2) \\
& \quad NF(T_2|p_1 \cdot p_2) = NF(T_2|p_2)
\end{aligned} \tag{4.2}$$

This formula can be easily extended to sequentially connect more than two simple paths and compute the corresponding NR and NF values. Consider the scenario where query Q is composed by $p_1 \cdot p_2 \cdot p_3 \ldots p_n$. We first calculate $NR(R_1|p_1 \cdot p_2)$

and $NF(T_2|p_1 \cdot p_2)$, then in turn compute the next NR and NF until we reach $NR(R_1|p_1{\cdot}p_2{\cdot}p_3{\dots}{\cdot}p_n)$ and $NF(T_n|p_1{\cdot}p_2{\cdot}p_3{\dots}{\cdot}p_n)$. In practice, if the simple path p_i contains more than two nodes, it would be decomposed into a set of basic paths which hold only two nodes. The NRs and NFs of parent-child paths can be directly retrieved from the estimation system. The processing of ancestor-descendant paths will be discussed later in this section.

Parallelizing Linear Paths

Given a set of simple paths $p_i, (1 \leq i \leq n)$ and the corresponding $NR(R_i|p_i)$, where R_i is the first node in path p_i, if $R_i = R_j = R(i{\neq}j; 1 \leq (i,j) \leq n)$, we can combine all p_i into a simple twig where R is the root node and every p_i is a branch. For example, the simple twig query $A[/B]/C$ is constructed from simple paths A/B and A/C.

There are two possible relationships among the branches in the path obtained: intersection and union. In the case of intersection, we need to estimate the size of R that appear in all of p_i. Otherwise, the path condition is satisfied if R is the root node of any one of the l_i. We denote the intersection and union associations in the paths as $p_1{\cap}p_2{\dots}{\cap}p_n$ and $p_1{\cup}p_2{\dots}{\cup}p_n$ respectively.

If query Q is given by $p_1{\cap}p_2{\dots}{\cap}p_n$, then the node ratio $NR(R|Q)$ is computed from the intersection of all p_i:

$$NR(R|Q) = \prod_{i=1}^{n} NR(R|p_i) \tag{4.3}$$

On the other hand, suppose Q is given by $p_1{\cup}p_2{\dots}{\cup}p_n$. Given the values of the corresponding node ratios $NR(R|p_i)$ where R is the root node of Q, we have following formula that is based on set theory:

$$NR(R|P) = \sum_{i=1}^{n} NR(R|p_i) - \sum NR(R|p_{i_1})NR(R|p_{i_2}) + ...+$$
$$(-1)^{(k-1)} \sum NR(R|p_{i_1})NR(R|p_{i_2})...NR(R|p_{i_k}) \tag{4.4}$$
$$(1 \leq (i_1, i_2...i_k) \leq n; i_a \neq i_b \ if \ a \neq b)$$

Ancestor-Descendant Basic Paths

Next, we compute the NR and NF of an ancestor-descendant basic path. The basic idea is that given an ancestor-descendant basic path, we try to "recover" its parent-child path expression based on the Basic Path Independence Assumption. This task can be achieved by scanning all basic parent-child paths stored in the system.

Algorithm 7 Recover Parent-Child Path Expression

Input: An ancestor-descendant path $Q = X//Y$
Output: Parent-child expression Q'

1. Set Q' to empty, add node X into Q' as the root node.
2. Scan all basic parent-child paths: if a path starts with X, attach the child node in the path as a child node of X in Q'.
3. For each new attached leaf node n (which is not Y) in Q', scan the basic parent-child paths and add the child node of a path if its parent node is the same as n.
4. Repeat step 3 until no new nodes can be added.
5. Delete root-to-leaf paths in Q' if the leaf node is not Y.

Algorithm 7 shows how to recover the parent-child path expression. Given an ancestor-descent query $Q = X//Y$, we generate a new query Q' which initially contains only one node X. Next, all basic parent-child paths are scanned and the child node of a paths is attached as a child node of X in Q' if the path starts with

node X. After that, we repeat this step for each newly attached leaf node (which is not a Y node) in Q' until no new node can be added into Q'. Finally, we delete all branch paths in Q' if their leaf nodes are not Y (the child node in the original ancestor-descendant query Q) and output Q'.

Example 4.11: Suppose we issue the query $Q = A//C$ over the XML instance in Figure 4.3. Q' initially contains one A node. After scanning all parent-child paths, Q' becomes $A[/B]/C$. Next, we check the paths to find those which start with node B. Finally, the output Q' is $A[/B/C]/C$. □

4.3.4 Estimation Algorithm

Based on the summary statistics and the various methods to aggregate them, we develop an algorithm to estimate the frequency of a target node in a given general path query. Our generalized estimation technique employs a bottom-up approach to compute the NR and NF of the nodes, before calculating node frequency in a top-down manner. This essentially implies that we first decompose a general query tree pattern into a set of basic parent-child paths, then recursively aggregate the NR and NF information using the methods described in Section 4.3.3.

We use a simple example below to explain the idea before presenting the details of the algorithm.

Example 4.12: A path query can typically be expressed as a linear (simple) path followed by a twig (see Figure 4.1(d)). Assume that we have a general query $Q_1 = l_1 \cdot t_1$ with target node T in t_1 and the common node between l_1 and t_1 is C_1. Obviously, $S_{Q_1}(C_1) = f(R_1) * NR(R_1|Q_1) * NF(C_1|Q_1)$ while R_1 denotes the root node of Q_1. After computing the selectivity of C_1, we can calculate the number of T as $S_{Q_1}(T) = S_{Q_1}(C_1) * NF(T|t_1)$. To obtain $NF(T|t_1)$, we need to get the NR

and NF values of the subtrees of t_1 first. As a result, we decompose query Q_1 into a set of sub-queries and collect their NR and NF values in a bottom-up manner. This leads to the design of Algorithm 8 □

Algorithm 8 calls a function $PathStat$ which returns an object s_q containing the values of NR and NF. Suppose Q is a general path query with root node *Root* and target node T. We call the path from *Root* to T the *primary path*. The variable NF *list*, initially set to empty, stores the NF values of the sub-paths in the primary path.

Lines 1-5 in function $PathStat()$ check whether T is contained in simple path portion of the current path Q. If it is, we decompose Q (in function $getTargetNode$ $NRNF()$, Algorithm 9) into a linear path l' (the path from R to T), and a general path g' (the subtree rooted at T). In order to obtain $NR(R|Q)$ and $NF(T|Q)$, we recursively call $PathStat$ to determine the NR and NF of l' and g' respectively. A special case occurs when Q is a linear path and T is the last node in Q; then g' is a single node. Note that this part (Lines 1-5) is only executed once although the function $PathStat()$ is recursively called.

In Lines 6-25 (Algorithm 8), current query Q may be decomposed or instantiated into a set of sub-paths depending on its type. To compute the NR and NF of Q, it is necessary to know the NR and NF of its sub-paths. Hence, $PathStat$ is invoked recursively (in the functions of Algorithm 9) to determine the NR and NF of every sub-path until a basic parent-child path or a single node is reached. The NR and NF of every sub-path is then aggregated to obtain the final result size. Note that only the NFs of nodes that occur in the primary path are stored in the NF *list* (lines 30-31) since the rest of the NF information does not contribute to target node selectivity estimation.

Example 4.13: Consider the general query Q in Figure 4.5 with target node N.

Algorithm 8 Estimate Query Selectivity

Input: Query Q with target node T.
Output: Estimated selectivity $S_Q(T)$

1: Initialize $NF\ list = \{\}$, $cursor = 0$
2: $s_q = PathStat(Q)$
3: $S_Q(T) = f(Root) * s_q.NR * \prod_{i=0}^{cursor} e_i(e_i \in NF\ list)$

Function $PathStat(p)$

1: **if** $isLinearTwig(Q)$ and T in linear part **then**
2: $s_q = getTargetNodeNRNF(Q)$
3: $e_0 = s_q.NF;$
4: return s_p
5: **end if**
6: **if** $isSingleNode(Q)$ **then**
7: $s_q = getSingleNodeNRNF(Q)$
8: return s_q
9: **end if**
10: **if** $isBasicParentChild(Q)$ **then**
11: $s_q = getParentNRNF(Q)$
12: return s_q
13: **end if**
14: **if** $isBasicAncestorDescendant(Q)$ **then**
15: $s_q = getAncestor(Q),$
16: return s_q
17: **end if**
18: **if** $isSimple(Q)$ **then**
19: $s_q = getSimple(Q)$
20: return s_q
21: **end if**
22: **if** $isTwig(Q)$ **then**
23: $s_q = geTwig(Q)$
24: return s_q
25: **end if**
26: **if** $isLinearTwig(p)$ **then**
27: $s_q = getLinearTwig()$
28: Let C be the common node between linear part and branch part
29: **if** $inPrimaryPath(C)$ **then**
30: $cursor++$
31: $e_{cursor} = s_q.NF$
32: **end if**
33: return s_p
34: **end if**

Algorithm 9 Estimate Query Selectivity (Cont)

Function $getTargetNodeNRNF(Q)$

1: Let R be the root node of Q
2: Let l' be the simple path from R to T and g' be the general query pattern rooted at T
3: $s_q.NR = NR(R|l' \cdot g')$, $s_q.NF = NF(T|l' \cdot g')$
4: /*Calculate NR and NF values of l' and g', then serialize them*/
5: return s_q

Function $getSingleNodeNRNF(Q)$

1: $s_q.NR = 1$, $s_q.NF = 1$
2: return s_q

Function $getParentNRNF(Q)$

1: $s_q.NR = NR(Q)$, $s_q.NF = NF(Q)$
2: /*$NR(Q)$ and $NF(Q)$ are retrieved from estimation system*/
3: return s_q

Function $getAncestor(Q)$

1: Let Q be the format $X//Y$
2: Get parent-child expression Q' by calling Algorithm 7
3: $s_q = PathStat(Q')$ with Y as target node
4: return s_q

Function $getSimple(Q)$

1: Decompose Q into a set of basic path $p_i(1 \leq i \leq n)$
2: Let R and T be the first and last node in Q respectively
3: $s_q.NR = NR(R|p_1 \cdot p_2 \cdot ...p_n)$, $s_q.NF = NF(T|p_1 \cdot p_2 \cdot ...p_n)$
4: /*The above values can be calculated by serializing basic paths*/
5: return s_q

Function $getTwig(Q)$

1: Decompose Q into a set of general paths $g_i(1 \leq i \leq n)$, every branch of Q is a g_i
2: Let R be the root node of Q
3: $s_q.NR = NR(R|g_1 \cap ...g_n)$, $s_p.NF = null$
4: /*The above values can be calculated by parallelizing paths, note the relationship between paths is intersection/
5: return s_q

Function $getLinearTwig(Q)$

1: Let l and t be the linear and twig parts of Q respectively
2: Let R and C be the first and last node of l respectively
3: /*note C is also the root node of t*/
4: $s_q.NR = NR(R|l \cdot t)$, $s_q.NF = (C|l \cdot t)$
5: /*Serializing l and t*/
6: return s_q

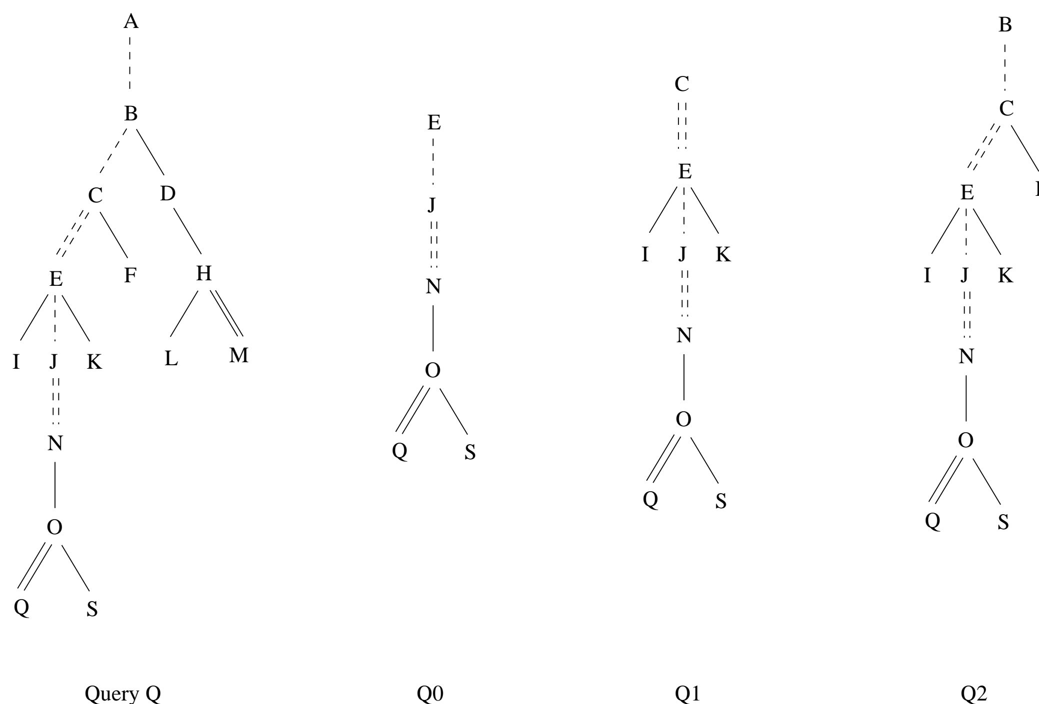


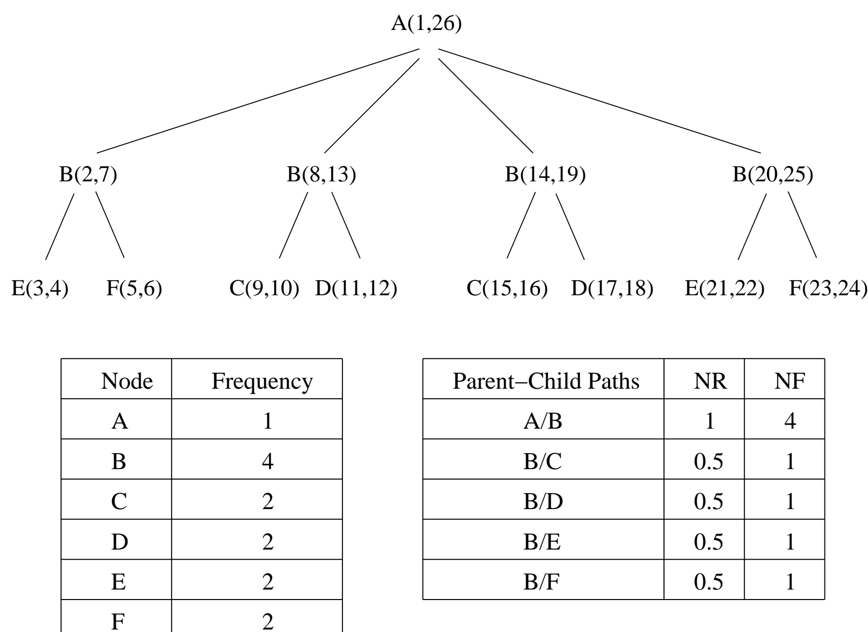
Figure 4.5: Estimating Frequency of Node N in Query Q

Then the path from A to N is the primary path (indicated by the dashed line in Q). Since N does not lie in the linear portion of Q, which is A/B, we recursively decompose Q until we obtain the path Q_0. We store $e_0 = NF(N|Q_0)$ in the *NF list*. On backtracking, we encounter path Q_1 which comprises a linear path and a twig which have node E in common. Since E lies in the primary path, we store node $e_1 = NF(E|Q_1)$. Similarly, we put $e_2 = NF(C|Q_2)$ and $e_3 = NF(B|Q)$ in the *NF list*. Finally, the e_i in the *NF list* are used to compute the frequency of node N in Q. That is, $S_Q(N) = f(A) * NR(A|Q) * e_3 * e_2 * e_1 * e_0$. □

4.4 Histogram-Based Estimation

The compact statistical approach introduced in the previous section works well when the data is regularly distributed. However, skewness in the data reduces

the accuracy of the estimation. Here, regular distribution and skewness mean whether the paths in XML dataset are distributed independently or correlatively respectively. In other words, in regularly distributed dataset, the Basic Path Independence Assumption will stand well. For example, in Figure 4.6, the child nodes C and D are not independently distributed under all the B nodes. This skewness could be due to the correlation between C and D. That is, C and D tend to occur together under the same B node. Suppose we issue a query $Q = B[/C]/D$ $(B/C \cap B/D)$ with target node B in this XML instance. Since $NR(B|Q) = NR(B|B/C) * NR(B|B/D) = 0.25$ according to the parallelization method, we have estimated $S_Q(B) = f(B) * NR(B|Q) = 1$, while the correct selectivity of node B is 2. This estimation error caused by data skewness becomes unacceptably high for Internet-scale XML data. In this part, we explore the solution of building histograms to reduce estimation error.



Node	Frequency
A	1
B	4
C	2
D	2
E	2
F	2

Parent−Child Paths	NR	NF
A/B	1	4
B/C	0.5	1
B/D	0.5	1
B/E	0.5	1
B/F	0.5	1

Figure 4.6: Example of a Skewed XML Instance and its NR-NF Values

4.4.1 Histogram Structure

The proposed histogram structure is based on the interval numbering scheme [57, 87]. This scheme facilitates the quick determination of the ancestor-descendant relationship between any two nodes in an XML document as shown in Figure 4.6. The basic idea behind is that the interval of a descendant node in a tree is contained in the intervals of its ancestor nodes. More details of interval based numbering scheme can be found in [57, 87].

We construct a set of buckets, or a *bucket set*, on selected parent-child basic paths to capture the underlying skewed data. The histogram puts "similar" data into the same bucket to reduce estimation error. A bucket stores the corresponding intra-bucket root node frequency, the NR value and the NF values.

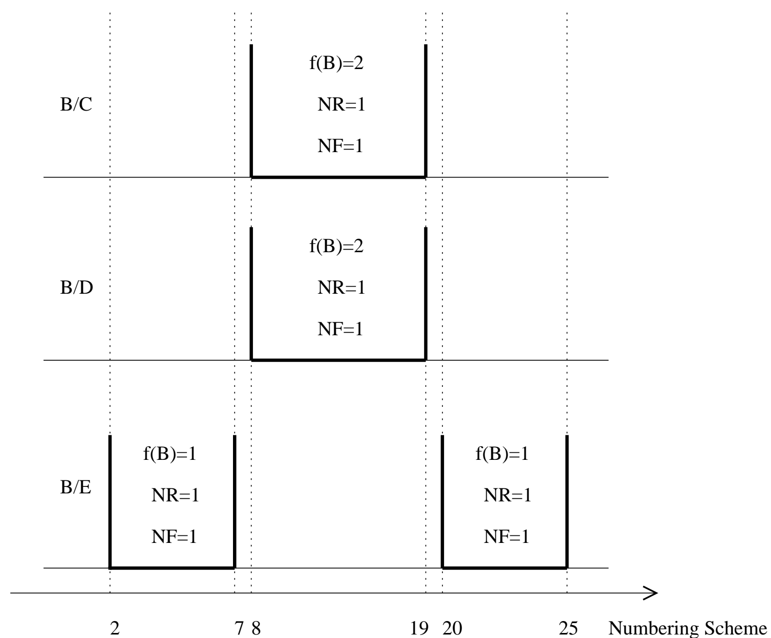
Example 4.14: Figure 4.7 shows the buckets that are built on the basic paths B/C, B/D and B/E for Figure 4.6. Each bucket contains the distribution of NR and NF in the range covered by the bucket. □

Histogram Construction

In order to decide whether a histogram should be constructed on a basic parent-child path P, we design an algorithm to detect the distribution of NRs and NFs of P based on the interval ranges where P occurs (see Algorithm 10).

We introduce two parameters to guide the construction of buckets: $Unit\ Factor$ (UF) and $Variance\ Factor$ (VF). UF controls the granularity of buckets. Let the interval range of a path P be $(P(s), P(e))$. Then the granularity of the bucket range is defined as $(P(e) - P(s))/UF$, and the range of every bucket built on path P must be a multiple of this granularity.

The variance factor VF indicates the difference of $NR(NF)$ between a bucket



Paths	NR	NF	Range
A/B	1	4	(1,26)
B/C	0.5	1	(8,19)
B/D	0.5	1	(8,19)
B/E	0.5	1	(2,25)
B/F	0.5	1	(2,25)

Overall Data

Paths	Root Frequency	NR	NF	Bounding
B/C	2	1	1	(8,19)
B/D	2	1	1	(8,19)
B/E	1	1	1	(2,7)
B/E	1	1	1	(20,25)

Buckets Data

Figure 4.7: Histograms of Paths

and the overall data. That is, if $NR(NF)$ in a range is greater than $avg(NR)*VF$ $(avg(NF)*VF)$ or less than $avg(NR)/VF$ $(avg(NF)/VF)$, we consider the data in this range as being significantly different from the average value, which will trigger the construction of a bucket to cover this range.

Algorithm 10 Tuning Buckets in Histogram

Input: Path P, VF, UF
Output: A set of buckets built on P

1: $unit = (P(e) - P(s))/UF$
2: /*$P(s), P(e)$ denote the start and end points of P*/
3: $B(s) = P(s)$
4: $B(e) = P(s)$
5: /*$B(s), B(e)$ denote the start and end point of a bucket*/
6: $bucket_status = 15$ (binary number '1111')
7: **while** $B(e) < P(e)$ **do**
8: $unit_status = getUnitStatus()$
9: **if** $unit_status > 0$ **then**
10: **if** $unit_status \cap bucket_status > 0$ **then**
11: $B(e) = B(e) + unit$
12: $bucket_status = unit_status \cap bucket_status$
13: **else**
14: output bucket $(B(s), B(e))$
15: $B(s) = B(e) + 1$, $B(e) = B(e) + unit$
16: $bucket_status = unit_status$
17: **end if**
18: **else**
19: output bucket$(B(s), B(e))$
20: $B(s) = B(e) + unit + 1$, $B(e) = B(e) + unit + 1$
21: $bucket_status = 15$
22: **end if**
23: **end while**

Algorithm 10 constructs the histogram on a path P by scanning all the interval information of the instances of P. The variables $unit_status$ and $bucket_status$ are represented by a 4-bit integer. Each bit indicates a status: high NR, high NF, low NR and low NF. In the procedure of scanning, if the status of a unit range is greater than 1, then the data distribution in this unit is significantly different from

the average value. If the data distribution in the next unit also differs significantly from the average value, then we check whether these two units have the same features, that is, whether both have high (low) NR or NF. This test can be carried out quickly by a bit AND operation. If the two units have common characteristics, then we can merge them into one bucket, and use the intersection of their status as the final bucket status. Otherwise, we simply output the first unit as a bucket and continue to process the next unit until we finish scanning the entire range of the path.

Note that the variables VF and UF actually control the bucket's depth and width respectively. Both of them directly affect memory usage. When UF increases or VF decreases, more buckets are constructed in memory.

Histogram Maintenance

An interval-based histogram can be updated by a two-step operation when the underlying data is modified. The details are given as follows.

1. Rebuild.

   Algorithm 10 is called to build new buckets. However, instead of using the entire range of the given path P, we only need to check the range where the data updates occur. For example, in Figure 4.7, if updates occur in the first bucket of path B/E, then we will only rebuild buckets by checking the range of this first bucket. On the other hand, if updates take place in the area between the first and second buckets of path B/E, then rebuilding would be carried out on this "empty" range. Note that an existing bucket may be deleted after rebuilding if the new data distribution values NR and NF inside the bucket are very close to the overall NR and NF values.

2. Merge.

After some new buckets are created, we then check whether these new buckets have adjacent buckets and they could be merged or not. The adjacent buckets will be merged to form a new bucket if the statistical information in the new bucket differs greatly from the average, which is determined by the variable VF in Algorithm 10.

4.4.2 Estimating XML Queries

When we estimate the selectivity of XML queries by using histogram data, we generate a new structure which is "compatible" to both the histograms of the paths needed to be merged (serializing or parallelizing). After that, the histogram of each path is transformed to the structure of this "compatible" bucket set, and their intra-bucket NR and NF values can now be aggregated directly.

Definition 4.3 (Compatible Bucket Set) *Given a bucket set A built on a basic parent-child path P_A, a bucket set B is compatible with A if all the following conditions hold.*

1. *Range covered by adjacent buckets in B must be consecutive.*
2. *Every bucket in B must be either totally contained in some bucket of A or totally outside the buckets of A.*
3. *Entire range covered by B contains the entire range of P_A.*

Example 4.15: Consider Figure 4.8(a) where a bucket set built on path P_2 is compatible with that built on path P_1. However, the bucket set built on path P_3 is not compatible with that built on P_1. Figure 4.8(b) shows two paths P_1, P_2 and a bucket set structure P_c that is compatible to both P_1 and P_2. Note that P_c has the minimum number of buckets and the minimum total interval range.

□

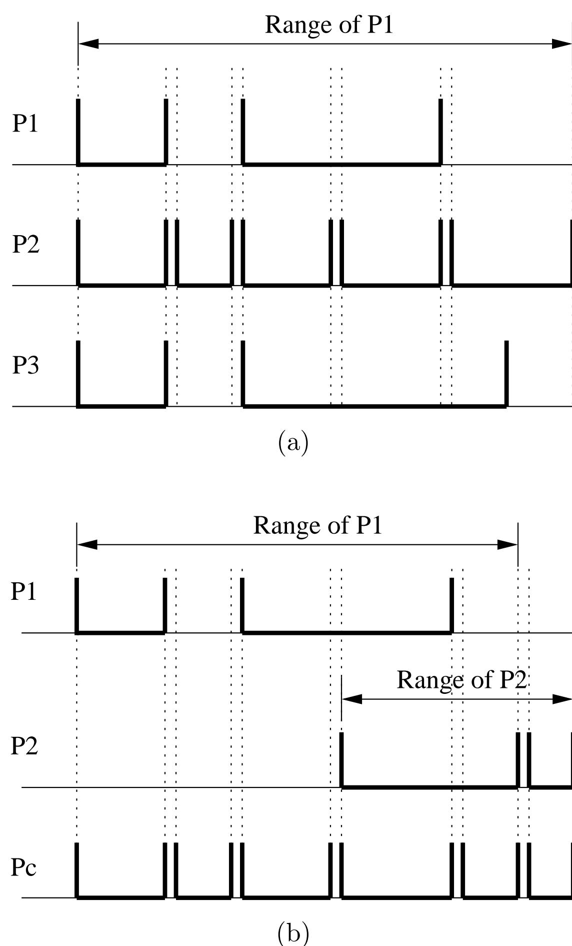


Figure 4.8: Compatible Bucket Sets

Let us now examine how a bucket set can be converted into a compatible bucket set. Suppose the bucket set $B\{b_1, b_2, ...b_n\}$ has been built on some path P where R is the root node of P, and let $b_i(rf)$, $b_i(s)$ and $b_i(e)$ indicate the root frequency, start edge and end edge of bucket b_i respectively while $P(s)$ and $P(e)$ denote the minimal and maximal values in the range covered by P. When bucket set B is converted to its compatible bucket set C, we compute the NR, NF and root node frequency inside every bucket c_j. For each c_j, we check whether it is contained in some b_i. If it is, based on the assumption of uniform distribution of

the intra-bucket data, we have NR and NF of c_j equal to those of b_i, and the $c_j(rf)$ is $b_i(rf) * (c_j(e) - c_j(s))/(b_i(e) - b_i(s))$. If c_j is not contained in any b_i but is contained in the interval $(P(s), P(e))$, we assume that the data outside the buckets is uniformly distributed. Thus we have the following formulae to compute NR, NF and root node frequency of c_j.

$$c_j(rf) = \frac{[f(R) - \sum_{i=1}^{n} b_i(rf)] * [c_j(e) - c_j(s)]}{P(e) - P(s) - \sum_{i=1}^{n} [b_i(e) - b_i(s)]}$$

$$c_j(NR) = \frac{f(R) * NR - \sum_{i=1}^{n} b_i(rf) * b_i(NR)}{f(R) - \sum_{i=1}^{n} b_i(rf)}$$

$$c_j(NF) = \frac{f(R) * NR * NF - \sum_{i=1}^{n} b_i(rf) * b_i(NR) * b_i(NF)}{f(R) * NR - \sum_{i=1}^{n} b_i(rf) * b_i(NR)}$$

If c_j falls outside the range $(P(s), P(e))$, then we set the root frequency, NR and NF to 0.

Algorithm 8 can be slightly modified to estimate queries using histogram data. We use the following example to explain:

Example 4.16: Consider the twig query $Q = A/B[/C/E]/D$ issued with target node C. We first decompose the query into a set of basic queries A/B, B/C, B/D and C/E. When aggregating the NR and NF values, we combine the paths C/E and B/C before aggregating the histogram data of $B/C/E$ with that of B/D. This procedure is repeated until we obtain $NR(A|Q)$ and $NF(C|Q)$ of every bucket. Finally, we summarize the estimated frequencies of C in all buckets and output the result. □

4.5 Experiments

In this section, we examine the accuracy of the proposed estimation approach on both real-world and synthetic datasets.

The datasets used are IMDB [3], Shakespeare's Plays (SSPlays) [1], DBLP [4] and a synthetic XML dataset generated by XMark [2]. The synthetic dataset is more skewed compared to the real datasets. Table 4.1 gives the characteristics of the various datasets. Attributes are omitted for simplicity.

Dataset	Size (MB)	♯(Dist Elements)	♯(Dist Basic Paths)	♯(Elements)
IMDB	1	12	11	26,045
SSPlays	7.5	21	38	179,690
DBLP	22	26	35	545,658
XMark20	20	74	98	319,815

Table 4.1: Characteristics of Datasets

4.5.1 NR-NF Estimation Method without Histogram

In this part, we check the performance of our proposed solution without building any histogram. That is, only the NR and NF values of a parent-child path are utilized to estimate query selectivity.

Summary Collection Time and Statistics Size

Table 4.2 shows the time taken to collect the summarized NR and NF values and the statistics size for each dataset. As we have expected, the information collection time is reasonably short since only one sequential scan is needed. When implementing our proposed method, we encode every distinct element tag and distinct path. Thus, the total NR-NF usage (without histograms) of the statistics

is given by:

$$NR - NF \;\; Usage \;\; = \;\; \sharp DistinctNodes * 8 \; + \; \sharp DistinctPaths * 12$$

where eight bytes are used to represent element ID and frequency, and 12 bytes are applied to represent path ID, NR value and NF value. From the table, we find that the summary sizes for all datasets are very small due to the highly compact NR and NF values we capture.

Dataset	Summary Collection Time	Statistics Size
IMDB	1.1 S	0.23KB
SSPlays	2.8 S	0.63KB
DBLP	8.2 S	0.63KB
XMark20	7.2 S	1.77KB

Table 4.2: Summary Collection Time and Statistics Size

Sensitivity Experiments

This set of experiments investigates the accuracy of our proposed solution on positive XML queries without building histograms. We generate a set of positive queries on both real and synthetic datasets. These queries comprise parent-child linear queries, ancestor-descendant linear queries and parent-child twigs. Table 4.3 shows the workload in this part.

Relative error is employed to measure estimation accuracy in this part. Table 4.4 shows the performance of using the proposed compact statistical information to estimate the queries.

Overall, memory usage for the statistical information is low (see Table 4.2), and estimation results for the queries on the real-world datasets (IMDB, SSPlays, DBLP) are basically very accurate since data in these datasets is typically regularly distributed. Even for the ancestor-descendant queries, our proposed method can accurately "recover" the original parent-child expressions. For example, Table 4.5

	Queries	Dataset
Q1	//PLAY//TITLE	SSPlays
Q2	//PLAY/ACT/PROLOGUE/SPEECH	SSPlays
Q3	//PLAY/ACT[/EPILOGUE/STAGEDIR]/TITLE	SSPlays
Q4	//PLAY/PERSONAE[/PGROUP]/TITLE	SSPlays
Q5	//Movie/Directed_By/Director	IMDB
Q6	//Cast/Actor/LastName	IMDB
Q7	//Movie//Genre	IMDB
Q8	//proceedings//editor	DBLP
Q9	//dblp/inproceedings/title	DBLP
Q10	//australia/item/description/text	XMark20
Q11	//open_auction/annotation//listitem	XMark20
Q12	//listitem/text/emph[/keyword]/bold	XMark20

Table 4.3: Query Workload

shows all the parent-child expressions generated by the ancestor-descendant query Q1 in the workload and the calculated NR and NF values for each path (root node and target node are "$PLAY$" and "$TITLE$" respectively). This result totally matches the real XML data, clearly showing the effectiveness of our proposed solution for regularly distributed XML data.

In contrast, the error rate for the synthetic dataset (XMark20) is considerably higher since the dataset is much more skewed compared to the real-life datasets. That is, the path-independence assumption does not properly stand for this synthetic dataset.

Comparative Experiments

We also implement *XSketch* [64] using f-stabilize method and compare its accuracy and memory usage with our method. We generate 100 positive queries on IMDB, SSPlays, DBLP and XMark20 respectively using the template described in [64], i.e., twig queries with linear branch paths, and the target node is the leaf node. Only the parent-child relationship is used here. The length of the query path ranges

Queries	Estimated size	Actual size	Error
Q1	1031	1031	0
Q2	12	12	0
Q3	5	4	25.0%
Q4	33	33	0
Q5	520	520	0
Q6	6886	6886	0
Q7	1475	1475	0
Q8	1892	1892	0
Q9	58077	58077	0
Q10	310	315	1.6%
Q11	2052	2148	4.5%
Q12	15	41	63.4%

Table 4.4: Error Rates

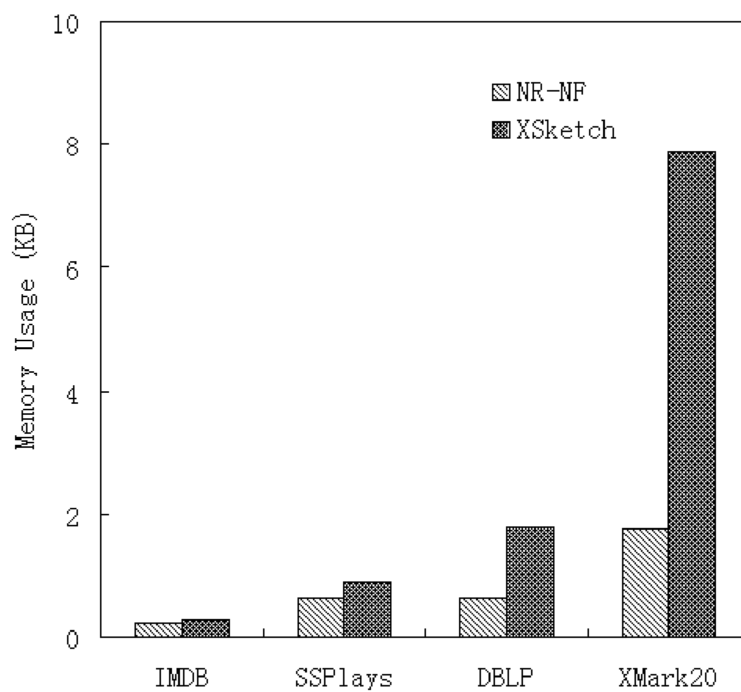
Parent-child path	NR	NF
PLAY/TITLE	1.0	1.0
PLAY/PERSONAE/TITLE	1.0	1.0
PLAY/ACT/TITLE	1.0	5.0
PLAY/ACT/SCENE/TITLE	1.0	20.216
PLAY/ACT/EPILOGUE/TITLE	0.162	1.0
PLAY/ACT/PROLOGUE/TITLE	0.324	1.0
PLAY/INDUCT/TITLE	0.054	1.0
PLAY/INDUCT/SCENE/TITLE	0.054	1.0
PLAY/PROLOGUE/TITLE	0.054	1.0

Table 4.5: Recovering Parent-Child Expressions for Query Q1

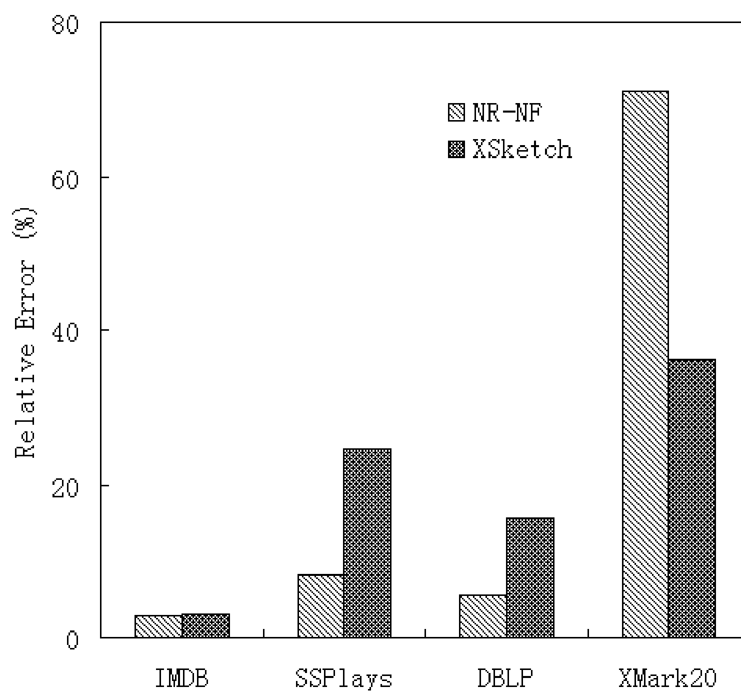
from three to 12 nodes.

Figure 4.9 shows the memory usage and average relative error of the two techniques for the various datasets. We observe that both techniques utilize a smaller amount of memory for the real-life datasets than for the synthetic dataset XMark20. This is largely because of the relatively regular structures of the real-life datasets. The regular data distribution in the real-life datasets also leads to a lower error rate for both methods compared to the synthetic dataset.

For the synthetic dataset, our approach requires a smaller memory footprint compared to *XSketch*. The latter requires more memory to achieve a lower relative



(a) Memory Usage



(b) Error Rates

Figure 4.9: Comparative Experiments

error. This is because *XSketch* uses a summarized graph to describe XML data. For skewed data, it naturally requires higher memory space consumption while yielding more accurate estimated results.

Scalability

In this part, we investigate the scale-up property of the *NR-NF* methods when file size varies. We run the positive queries generated previously on three XMark [2] datasets with various file sizes (Table 4.6). Table 4.7 shows that relative error is barely affected by increasing file size. At the same time, there is no increase in memory usage. This result is expected since varying the sizes of XML files does not affect the number and distribution of basic parent-child paths.

Dataset	Size (MB)	♯(Dist Elements)	♯(Dist Basic Paths)	♯(Elements)
XMark20	20	74	98	319,815
XMark40	42	74	98	639,178
XMark60	63	74	98	959,495

Table 4.6: Characteristics of Datasets for Scalability Test

Dataset	Sum Collection Time	Statistics Size	Error Rate
XMark20	7.2 S	1.77KB	71.0%
XMark40	27 S	1.77KB	68.3%
XMark60	1 Min 54 S	1.77KB	70.2%

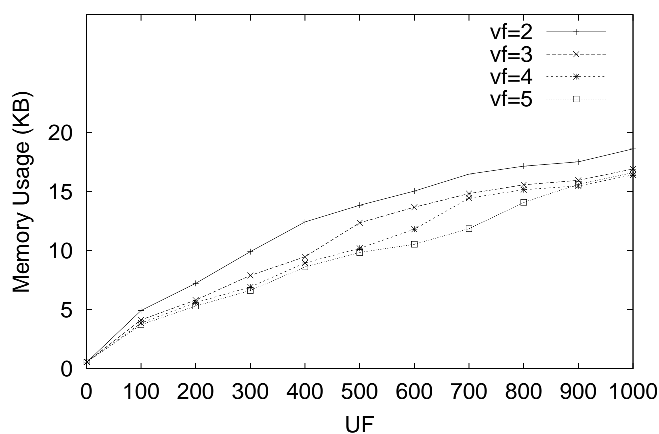
Table 4.7: Summary Collection Time, Statistics Size and Error Rate for Scalability Test

4.5.2 NR-NF Estimation Method with Histograms

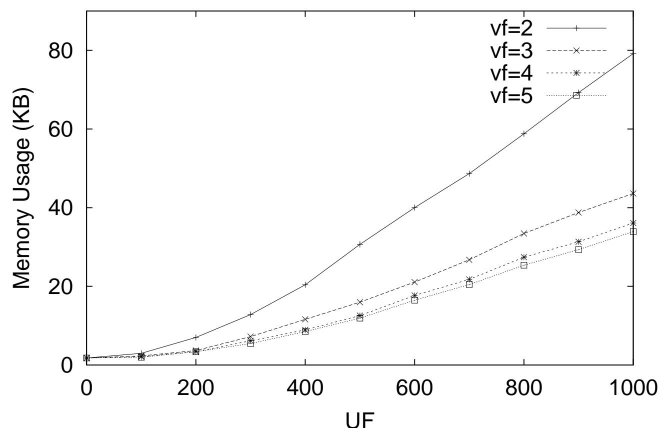
In this part, we examine the evaluation performance of the *NR-NF* approach with histogram data.

Statistics Size and Sensitivity Experiments

This set of experiments investigates the accuracy of the proposed histogram solution with various statistics sizes by using the 100 queries generated in Section 4.5.1 for each dataset.



(a) DBLP



(b) XMark20

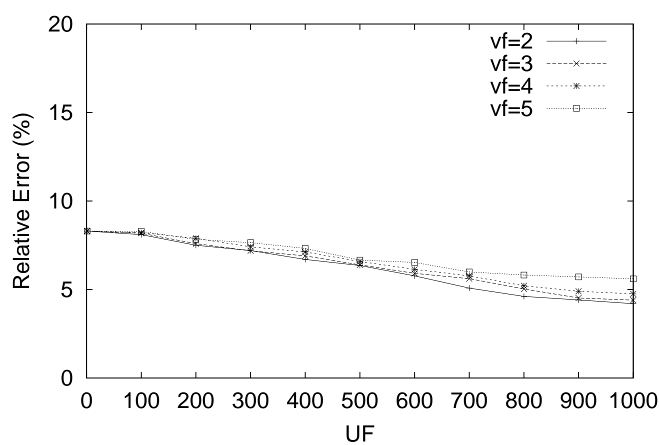
Figure 4.10: Memory Usage with Histograms

Figure 4.10 shows total memory usage for the DBLP and XMark20 datasets under varying UF and VF values. The same UF and VF values, e.g., 100 and 3 respectively, are used to scan data of each distinct path, and buckets are constructed when data skewness is detected.

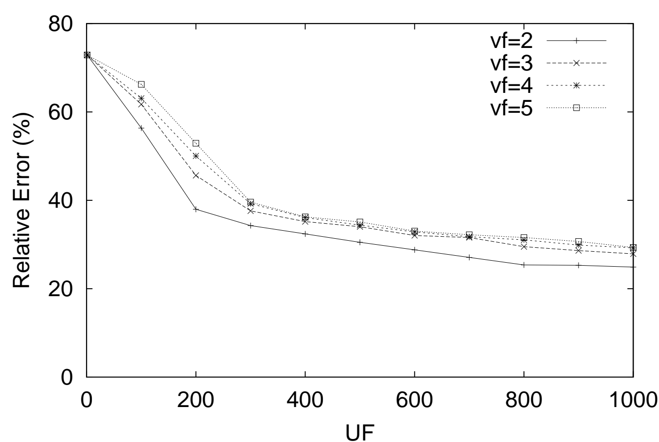
Total memory usage is given by the summary of the NR-NF and histogram

sizes. For histograms, each bucket is a 24-byte tuple with intra-bucket data (ID, root frequency, NR, NF, range.start, range.end).

$$Total\ Memory\ Usage = NR - NF\ Usage + Histogram\ Usage$$
$$Histogram\ Usage = \sharp Buckets * 24$$



(a) DBLP



(b) XMark20

Figure 4.11: Error Rates with Histograms

As expected, the memory usage of our histogram structure grows as UF increases (Figure 4.10). This is because the value of UF determines the unit width of buckets. The larger the UF value (the smaller the unit width) is, the larger the

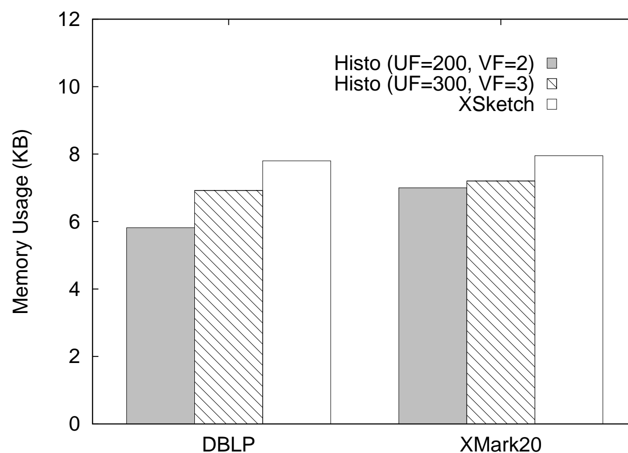
number of buckets is possibly obtained. On the other hand, the VF parameter determines the "distance" between the skewed data and the overall data. The lower the value of VF is, the more data would be detected as "skewness" in the XML files. We observe that UF value has a greater influence on memory requirement than VF value does. This implies in this histogram structure, bucket width is the major factor affecting memory space consumption.

Figure 4.11 shows the selectivity estimation errors of queries. There is a corresponding reduction in the relative error with the increasing memory usage. In addition, it can be observed that our proposed technique utilizes a smaller amount of memory for DBLP dataset compared to that required by the XMark under the same UF and VF values (see Figure 4.10). This is largely because of the relatively regular structure of the real-world dataset. This regular data distribution in the DBLP dataset also leads to a lower error rate (Figure 4.11).

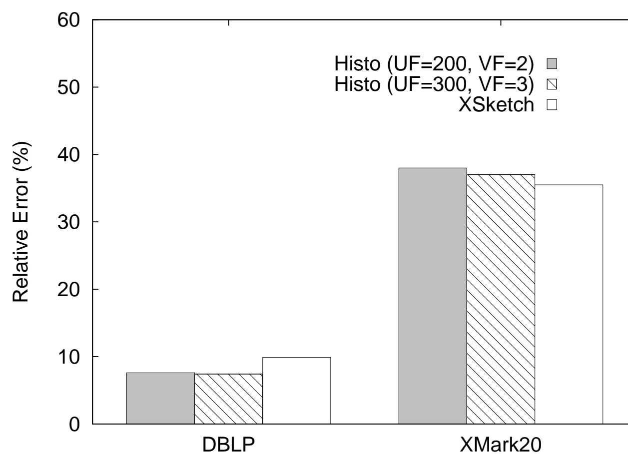
Comparative Experiments

We compare the histogram-based approach with the f-stabilize $XSketch$ solution. Since the size of the overall NR-NF estimation information of our approach is much smaller than that of $XSketch$, we choose the UF and VF values which consume roughly same memory space as $XSketch$. The comparison result is shown in Figure 4.12. We observe that our solution has similar values in both memory usage and error rate when UF and VF are (200,2) and (300,3) respectively. The XMark dataset is skewed and contains more irregularity than the real-world datasets. $XSketch$ explicitly stores the graph structure of the XML data, and requires more memory space when the underlying data is irregular. In contrast, the proposed histogram-based method captures the two variables of basic parent-child paths, which are independent of the structural complexity of the XML graph. This

leaves us additional memory space to build histograms to capture data distribution, leading to improved estimation accuracy.



(a) Memory Usage



(b) Error Rates

Figure 4.12: Histogram-Based Approach vs. *XSketch*

Statistics Collection Time

In this experiment, we evaluate the cost of constructing summary information for selectivity estimation. Table 4.8 shows the statistics collection time of the proposed histogram solution and *XSketch* [64].

We observe that our proposed method requires much less time to construct the

Solution	Statistics Size	Construction Time
DBLP (UF=200, VF=2)	5.82 KB	7.8 S
DBLP (UF=300, VF=3)	6.92 KB	8.1 S
DBLP (XSketch)	7.80 KB	42 S
XMark20 (UF=200, VF=2)	7.00 KB	12.4 S
XMark20 (UF=300, VF=3)	7.21 KB	14.8 S
XMark20 (XSketch)	7.95 KB	1 min 3 S

Table 4.8: Statistics Construction Time

statistical information than $XSketch$ does. This is because for histogram construction, only one sequential scan on the interval based NR and NF values is required as introduced in Algorithm 10. Considering the fact that these detailed NR and NF values are already available in the system (disk-based), which are obtained when calculating the NR and NF values of the XML dataset first time, this result is expected.

In contrast, [64] shows that building an optimal $XSketch$ model is an *NP*-hard problem. It utilizes a greedy refinement strategy to incrementally add statistics on the existing summary information. As a result, the construction time grows quickly when the statistics size increases. In Table 4.8, the worst case happens for the XMark dataset with the statistics size of 7.95 KB. In addition, this problem becomes more severe if we further increase the size of statistics. In the experiments of Chapter 5, we show that the construction time even reaches more than one week with 90 KB statistics.

Moreover, our proposed solution uses simple structures to store statistical information, which is obviously easier to implement and maintain than $XSketch$. In contrast, $XSketch$ utilizes complex data structure and thus the statistical information is more difficult to store and update.

4.6 Conclusion

In this chapter, we have presented a comprehensive solution for estimating the result sizes of general XML query patterns. Our system captures highly concise information, NR and NF, for every distinct parent-child path in an XML dataset. We have described how the statistical information can be aggregated to estimate XML query sizes. For skewed XML data, we design interval-based histograms to capture the underlying data distribution. Experiments on both real-world and synthetic datasets indicate that our proposed method is able to achieve very low error rates with a small amount of memory for regularly distributed real-world XML datasets, and our histogram-based method can increase estimation accuracy with minimal additional memory requirement. In addition, our proposed solution uses simple data structures to store the statistical information, which is easy to implement and efficient on statistics construction.

CHAPTER 5

A Path-Based Selectivity Estimator for XPath Expressions with Order Axes

5.1 Introduction

XML is a model of ordered tree that specifies the sequence order of sibling nodes. The XML query language XPath [10] also supports order-based axes. In XPath [10], the *preceding* and *following* axes describe the sequence of nodes before/after the context node (excluding any ancestors/descendants). For example, the query "*//Storm/following::Tornado*" requires that the element *Tornado* must occur after the element *Storm*. In addition, XPath provides the *preceding-sibling* and *following-sibling* axes to select all preceding or following sibling nodes.

The existing labeling scheme solutions preserve the order information of XML data, such as interval based solutions [57, 87] and prime number approach [82]. The work in [75] examines how ordered XML can be stored and queried using a relational database system. However, a key issue that has been neglected in the literature is how the selectivity of XML queries with order-based axes can be estimated.

The selectivity estimation of XML queries with order axes is a challenging task, given the huge volume of order information that needs to be summarized. The existing XML selectivity estimators [15, 26, 58, 62, 65, 66, 83] are designed specifically for XML queries without order axes, and order information is typically not captured.

In this chapter, we describe a framework for estimating the selectivity of XPath expressions with order-based axes. To the best of our knowledge, this is the first work to address the problem of summarizing order information in XML data. The key contributions are as follows:

1. We use a path encoding scheme to aggregate the path and order information of XML data. This scheme associates each node in an XML tree with a path id that indicates the type of the path where the node occurs. Based on the path ids, the frequencies of element tags and sibling node sequences can be collected.

2. We design two compact structures, the *p-histogram* and the *o-histogram*, to summarize the path information and order information of XML data respectively. In order to reduce the effect of data skewness in the buckets, we use intra-bucket frequency variance to control the histogram construction. We also devise efficient heuristic algorithms for maintaining the histograms.

3. We develop methods to estimate the selectivity of XPath expressions. We first remove the irrelevant path ids associated with elements involved in a query. Then the frequency values of the remaining path ids are utilized to calculate the selectivity.

4. We carry out an extensive experimental study of the proposed approach on various real-world and synthetic datasets. The results show that the proposed

solution produces very low estimation error rates for XPath queries even with limited memory space.

The rest of this chapter is organized as follows. Section 5.2 describes the path and order information captured. The estimation methods for queries without and with order axes are introduced in Section 5.3 and 5.4 respectively. We present the data structure used in Section 5.5. Section 5.6 shows the experimental results. We conclude in Section 5.7.

5.2 Capturing Path and Order Information

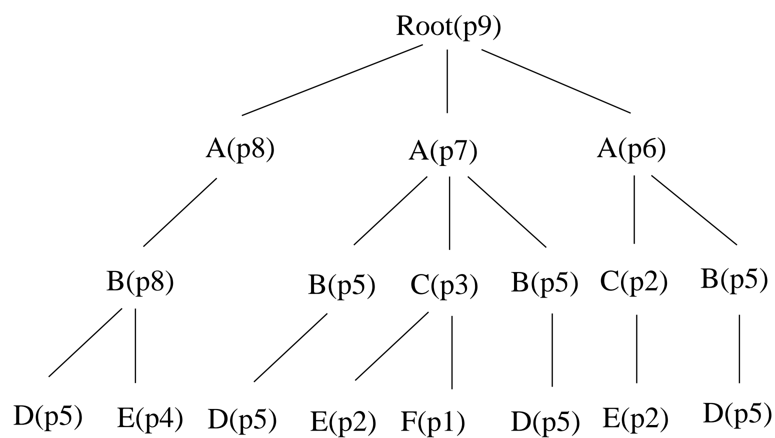
We use the path encoding scheme developed in Chapter 3 to label XML nodes, that is, each element node is associated with a path id which indicates the path type where the node occurs. The following example explains:

Example 5.1: In Figure 5.1, the path id of the first leaf node D in the XML instance is $p5$ (1000) since the encoding of the path "*Root/A/B/D*" on which D occurs is 1. The path id of the first C node (annotated with $p3$(0011)) is obtained by a bit-or operation on the path ids of its child nodes E and F, whose path ids are $p2$(0010) and $p1$(0001) respectively. All bit sequences are collected in a *path id table* (Figure 5.1(c)). □

In our proposed estimation system, the path and order information of XML data are captured in the *PathId-Frequency Table* and *Path-Order Table* respectively.

PathId-Frequency Table. Each tuple in the *pathId-frequency table* represents a distinct element tag in an XML document, and we aggregate all the path ids and their corresponding frequencies of each element tag.

Example 5.2: Figure 5.2(a) shows the *pathId-frequency table* for the XML data



(a) XML Instance

Root–to–leaf	Encoding
Root/A/B/D	1
Root/A/B/E	2
Root/A/C/E	3
Root/A/C/F	4

(b) Encoding Table

Bit–Seq	Int
0001	p1
0010	p2
0011	p3
0100	p4
1000	p5
1010	p6
1011	p7
1100	p8
1111	p9

(c) Path Id Table

Figure 5.1: Path Encoding Scheme

in Figure 5.1. Since there are two C elements occurring in the XML document, one of which is associated with path id $p3$ and the other one with $p2$, the entry for C in the *pathId-frequency table* comprises the set $\{(p2, 1), (p3, 1)\}$. □

Ele	(Path_id, Frequency)
A	(p6,1) (p7,1) (p8,1)
B	(p8,1) (p5,3)
C	(p2,1) (p3,1)
D	(p5,4)
E	(p4,1) (p2,2)
F	(p1,1)
Root	(p9,1)

(a) PathId – Frequency Table

Ele	p5	p8	Path Id
Root			Ele+
....			
C	2		
B			
A			
Root			+Ele
....			
C	1		
B			
A			

(b) Path–Order Table for Element B

Figure 5.2: Path and Order Information

Path-Order Table. We observe that the order information of XML elements is related to the types of the paths where the elements occur. Since the path ids of elements represent their path types, we captures the sibling order information based on the path ids of elements and these order information is stored in the *path-order table*. In this estimation system, each distinct element tag is associated with a *path-order table*.

Given an element tag X, each column in its *path-order table* denotes one path id on which the elements X occur, and each row represents one element tag in the XML document. There are two regions in the *path-order table*, namely, $+element$ and $element+$ regions. In the $+element$ region, a grid cell, denoted by $g(pathid, tag)$, represents the number of elements X with $pathid$ occurring before elements tag.

In contrast, the grid cell $g(pathid, tag)$ in *element+* area denotes the frequency of X with *pathid* occurring after elements *tag*.

Example 5.3: Figure 5.2(b) shows the *path-order table* for element B in Figure 5.1. There are totally four B elements, but only three out of them have sibling nodes. Since one B element annotated with $p5$ (see Figure 5.1) occurs before C, the cell ($p5$, C) in the *+element* area of the table is marked with 1. Similarly, two B elements with $p5$ occur after element C, thus the values in the corresponding cells ($p5$, C) in area *element+* is 2. And all the other cells are empty. □

Note that if an element with tag X occurs both after and before elements with tag Y, then the two rows for tag Y in the *path-order table* of X will count this X element.

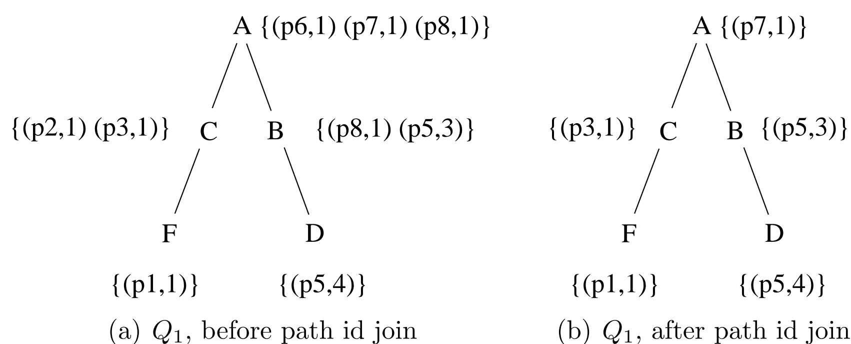
The *pathId-frequency table* captures all the path ids of each element tag and their frequencies. This information is utilized to estimate the selectivity of XPath expressions without order axes (see Section 5.3). The *path-order table* aggregates the frequencies of the path ids of the sibling nodes. This information is used to estimate the selectivity of XPath expressions with order axes (see Section 5.4). In addition, we design succinct data structures, *p-histogram* and *o-histogram*, to summarize the data in *pathId-frequency table* and *path-order table* respectively (see Section 5.5).

5.3 Estimating Selectivity of Queries with No Order Axes

This section describes our method of estimating simple and branch queries without order axes. The method is based on a path id join algorithm.

5.3.1 Path Join

Given an XPath query Q, path join first retrieves a set of path ids and the corresponding frequencies for each element tag in Q from the *pathId-frequency table*, then for each pair of adjacent element tags in Q, we use a nested loop to determine the containment of the path ids in their sets. Path ids that clearly do not contribute to the query result are removed. The frequency values of the remaining path ids are utilized to estimate query size. Details of the path join algorithm can be found in Chapter 3.



(a) Q_1, before path id join (b) Q_1, after path id join

Figure 5.3: Example of Path Id Join

Example 5.4: Consider query $Q_1 = //A[/C/F]/B/D$ in Figure 5.3(a) issued on the XML data in Figure 5.1, where the sets of path ids have been associated with the corresponding nodes. The path id join algorithm evaluates the query Q_1 by removing irrelevant path ids from the nodes in the query. The final result is shown in Figure 5.3(b). We observe that path id $p2$ for C is removed from the path id list because $p2$ cannot contain the path id $p1$ for F. Further, path id $p6$ and $p8$ for A are removed since they cannot contain the path id $p3$ for the child node C, etc. □

Given an XPath query Q with target node n (the node whose selectivity is to be estimated), we denote the selectivity for n as $S_Q(n)$ and the sum of frequencies of the remaining path ids after the path id join for n as $f_Q(n)$.

5.3.2 Estimating Simple Queries

A simple XPath query Q is of the basic form $/n_1/n_2/.../n_i$ where / and n_i denote the child axis and label of node respectively [1]. The following theorem computes the selectivity of simple XPath queries:

Theorem 5.1 *After path id join is applied on a simple query Q, the summarized frequency value $f_Q(n)$ of node n is the same as the selectivity $S_Q(n)$, that is:*

$$S_Q(n) = f_Q(n) \quad (5.1)$$

Proof: *After a path id join, only the path ids that satisfy the path id containment relationship are associated with the nodes involved in the query. Therefore, $f_Q(n)$ must be equal to $S_Q(n)$.* □

Example 5.5: Consider the query "$//A//C$" issued on the XML instance in Figure 5.1. After a path id join, the sets of path ids that are associated with A and C are $\{p6, p7\}$ and $\{p2, p3\}$ respectively. From the corresponding frequencies, we know that the selectivity for both A and C are 2. □

5.3.3 Estimating Branch Queries

We define a branch query pattern Q as $/n_1/.../n_i[/n_{i1}/.../n_{il}]/n_{i+1}.../n_m$. The path $/n_1/.../n_i$ is the trunk part while the paths $/n_{i1}/.../n_{il}$ and $/n_{i+1}.../n_m$ are the branch parts of the query. XPath provides different formats, such as $q_1[/q_2]/q_3$ or $q_1[/q_2][/q_3]$ to specify the position of the target node whose selectivity is to be estimated. In this work, we standardize the branch query pattern as $q_1[/q_2]/q_3$ where q_i is a simple query, and explicitly specify the target node.

[1] The relationship between nodes is not necessarily a parent-child relationship

Given a branch query $Q = q_1[/q_2][/q_3]$, if the target node n occurs in the trunk part q_1 of the query, then according to Theorem 5.1, we have $S_Q(n) = f_Q(n)$. However, if n occurs on the branch part q_2 or q_3 of Q, then $f_Q(n)$ may over-estimate $S_Q(n)$. This is because path ids are designed to directly capture the parent-child and ancestor-descendant containment relationship, but not the relationship between sibling nodes.

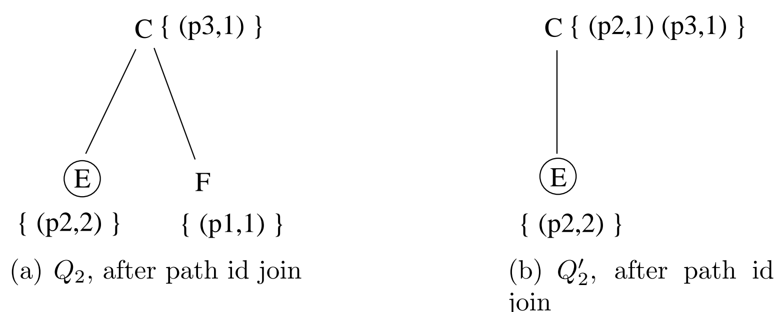


Figure 5.4: Estimating Selectivity of Branch Query

Example 5.6: Figure 5.4(a) shows a branch query $Q_2 = //C[/E]/F$ issued on the XML instance in Figure 5.1. The target node E is circled. The path id join associates node E with a path id set $\{(p2, 2)\}$. Figure 5.1 shows that only one E element with path id $p2$ is the answer. The other E element is not in the result because the path id $p2$ of its parent C has been removed during the path id containment test between C and F. Note the estimation result for C is in fact the exact answer. □

Example 5.6 shows that if the target node n is in the branch part of a query Q, $f_Q(n)$ may over-estimate the selectivity. However, if the target node n occurs in the trunk part of Q, then $f_Q(n)$ is the correct selectivity value. To compensate for this over-estimation, we devise a method which utilizes the correct selectivity information of other nodes to determine the selectivity of the target node occurring in the branch parts. This method is based on the following assumption:

- **Node Independence Assumption:** Given a branch query $Q = q_1[/q_2]/q_3$ with target node n in the branch part $q2$, the distribution of node n on q_2 in the XML data is independent of the distribution of all other nodes in the other branch path, i.e., q_3, on which n does not occur.

Example 5.7: Suppose we issue two queries $Q1 = //A[/B]/C$ and $Q2 = //A/B$ on the same XML document. Based on the Node Independence Assumption, we have $S_{Q1}(B)/S_{Q1}(A) \approx S_{Q2}(B)/S_{Q2}(A)$ since the distribution of B under A is independent of the distribution of node C which occurs on the other branch. □

We now discuss the estimation method for branch queries. Suppose the target node n belongs to the branch q_2 of Q. To estimate $S_Q(n)$, we generate a simple query $Q' = q_1/q_2$ from Q by ignoring the branch q_3. The results of n_i (the last node of q_1) on query Q is a subset of that of n_i on Q' since Q has an additional branch q_3. Correspondingly, the results of nodes occurring on q_2 of Q also decrease. Based on the Node Independence Assumption, we can infer that $S_Q(n)/S_Q(n_i) \approx S_{Q'}(n)/S_{Q'}(n_i)$. Since Q' is a simple query, we can obtain the correct selectivity values of n and n_i in Q' based on Theorem 5.1, i.e., $S_{Q'}(n)= f_{Q'}(n)$ and $S_{Q'}(n_i) = f_{Q'}(n_i)$. In addition, $S_Q(n_i) = f_Q(n_i)$ since n_i is in the trunk part of Q. Thus, we obtain the following formula to calculate $S_Q(n)$:

$$S_Q(n) \approx f_{Q'}(n) * f_Q(n_i)/f_{Q'}(n_i) \tag{5.2}$$

Example 5.8: We continue from Example 5.6. Consider the query Q_2 in Figure 5.4(a) where node C is the last element node in the trunk part (which corresponds to n_i in Formula 5.2), and E is the target node n. We generate a new query Q'_2 by cutting off the branch path where the target node does not occur (see Figure 5.4(b)). After a path id join on both queries, the values of $f_{Q_2}(C)$, $f_{Q'_2}(C)$ and $f_{Q'_2}(E)$ are 1,

2 and 2 respectively. Therefore, we estimate $S_{Q_2}(E)$ as $f_{Q'_2}(E) * f_{Q_2}(C)/f_{Q'_2}(C) = 1$.

$\square$

The error bounds of the proposed estimation method if the target node n occurs in the branch part is $[f_Q(n_i), f'_Q(n)]$, where n_i is the last element label in the trunk part q_1 of Q. The upper bound of $S_Q(n)$ is $f_{Q'}(n)$ because $f_Q(n_i)$ is always smaller than or equal to $f_{Q'}(n_i)$ in Formula 5.2. The lower bound of $S_Q(n)$ is $f_Q(n_i)$, which occurs when each node instance n_i has only one descendant n ($f_{Q'}(n) = f_{Q'}(n_i)$).

5.4 Estimating Selectivity of Queries with Order Axes

Building on the techniques described in the previous section, we now discuss the selectivity estimation of queries with order axes. We first present the techniques to estimate queries with *preceding-sibling* and *following-sibling* axes, and then discuss how to extend these methods to estimate queries with *preceding* and *following* axes.

5.4.1 Preceding-Sibling/Following-Sibling Axis

An XPath query with order axes can be denoted as $\vec{Q} = q_1[/q_2/folls :: q_3]$ (or $q_1[/q_2/pres :: q_3]$) where $folls$ ($pres$) represents a *following-sibling* (*preceding-sibling*) axis. $\vec{Q}$ requires that both branches q_2 and q_3 occur under q_1, and the entire path expression q_2 happens before (after) q_3 in an XML instance. We denote the counterpart query without order axes of $\vec{Q}$ as $Q = q_1[/q_2]/q_3$. Q can be generated from $\vec{Q}$ by removing the order axes in $\vec{Q}$.

Recall that the *path-order table* stores the sequence information of sibling nodes. Hence, given an XPath query $\vec{Q} = q_1[/q_2/folls :: q_3]$, we can use the *path-order table* to compute the selectivity of sibling nodes in $\vec{Q}$, i.e., the first nodes of q_2 and

q_3. The results of these sibling nodes are then utilized to compute the selectivity of the other nodes occurring in the query.

The estimation method to determine the selectivity of the target node depends on whether the target node occurs in the branch part or the trunk part of the query.

Case 1: Target Node in Branch Part

We first consider the situation where the target node on the branch part is also a sibling node. Then we extend the estimation method to handle queries where the target node is in the branch part but is not a sibling node.

Consider the query $\vec{Q} = q_1[/q_2/folls :: q_3]$ with target node n_{i+1} (the first node of q_3). Path id join is first applied on $\vec{Q}$ (or $Q = q_1[/q_2]/q_3$). Next, if we directly use the remaining path ids of n_{i+1} to retrieve the frequency values from its *path-order table*, we may over-estimate the selectivity. This is because in the *path-order table* for n_{i+1}, there is no path id condition imposed on the element n_{i1} (the first node of q_2, the sibling node of n_{i+1}). However, in the query $\vec{Q}$ (or Q), we require that n_{i1} must occur in the query pattern q_1/q_2. To overcome this problem, we make the following assumption:

- **Node Order Uniformity Assumption:** Given m elements X such that they are the sibling nodes of Y and m_s out of m X elements occur before (or after) Y, these m_s X elements are uniformly distributed in all m X elements. That is, if we randomly select m' from m X elements, there will exist m'_s X elements which occur before (or after) Y, such that $m'/m \approx m'_s/m_s$.

We generate a simplified query $\vec{Q'}$=$q_1[/n_{i1}/folls :: q_3]$ from $\vec{Q}$ by deleting the branch part q_2 except for its first node n_{i1}. Then we compute the selectivity $S_{\vec{Q'}}(n_{i+1})$ and the selectivity of nodes in its counterpart Q'= $q_1[/n_{i1}]/q_3$ without order axes.

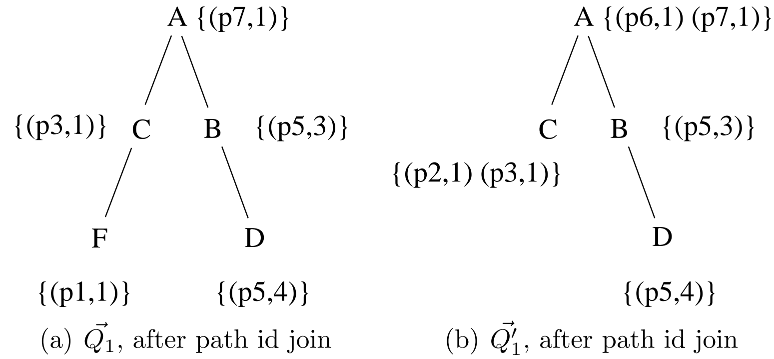
For query $Q' = q_1[/n_{i1}]/q_3$, we have $S_{Q'}(n_{i+1})$ (which is m in the Node Order Uniformity Assumption) elements tagged with n_{i+1} that satisfy Q' and are siblings of n_{i1}, where $S_{\vec{Q}'}(n_{i+1})$ (which is m_s) elements tagged with n_{i+1} occur after n_{i1}. If we select $S_Q(n_{i+1})$ (which is m') elements from the results of n_{i+1} on Q', we will have $S_Q(n_{i+1})/S_{Q'}(n_{i+1}) \approx S_{\vec{Q}}(n_{i+1})/S_{\vec{Q}'}(n_{i+1})$ ($S_{\vec{Q}}(n_{i+1})$ is m'_s). Thus, we get:

$$S_{\vec{Q}}(n_{i+1}) \approx S_{\vec{Q}'}(n_{i+1}) * S_Q(n_{i+1})/S_{Q'}(n_{i+1}) \tag{5.3}$$

The selectivity values $S_Q(n_{i+1})$ and $S_{Q'}(n_{i+1})$ can be estimated using the estimation method for branch queries (Section 4.3). The correct $S_{\vec{Q}'}(n_{i+1})$ can be retrieved from the *path-order table* for n_{i+1} as follows: After the path id join on Q', for each remaining *pid* associated with n_{i+1}, we retrieve $g(pid, n_{i1})$, i.e., the number of n_{i+1} elements with path id *pid* that occur after n_{i1} from the *path-order table* for n_{i+1}. Then the summary of all such $g(pid, n_{i1})$ is the selectivity of n_{i+1} for $\vec{Q}'$ according to the definition of *path-order table*.

Note that the value $S_{\vec{Q}'}(n_{i+1})$ obtained is the accurate selectivity of n_{i+1} in $\vec{Q}'$. This is because after path id join, the path ids associated with n_{i+1} represent the correct result set for n_{i+1} in the simple query q_1/q_3. In addition, the correct frequency value for n_{i+1} with these path ids which occur after n_{i1} are recorded in the *path-order table* for n_{i+1}. As a result, the retrieved value must be the correct $S_{\vec{Q}'}(n_{i+1})$.

Example 5.9: Figure 5.5(a) shows a query $\vec{Q_1}$=$A[/C[/F]/folls :: /B/D]$ with target node B. The simplified query $\vec{Q}'_1 = A[/C/folls :: B/D]$ is shown in Figure 5.5(b). The nodes in Figure 5.5 are annotated with the remaining path ids after a path id join. The value of $S_{\vec{Q}'_1}(B)$, which is 2, is retrieved from the *path-order table* for B (see Figure 5.2(b)) with element tag C and path id $p5$ which is the remaining path id in Figure 5.5(b). The values of $S_{Q_1}(B)$ and $S_{Q'_1}(B)$ are estimated as 1.3



(a) $\vec{Q}_1$, after path id join (b) $\vec{Q}'_1$, after path id join

Figure 5.5: XPath Query with Order Axes

and 2.6 respectively by using the estimation method without order axes. Finally, $S_{\vec{Q}_1}(B) = S_{\vec{Q}'_1}(B) * S_{Q_1}(B)/S_{Q'_1}(B) = 2 * 1.3/2.6 = 1$. □

Next, we consider the query $\vec{Q}$ with the target node n where n occurs in the branch part but is not the sibling node n_{i1} or n_{i+1}. Suppose that n occurs in q_3. In order to utilize the selectivity of n_{i+1} to estimate the selectivity of n, we make the following assumption:

- **Node Containment Uniformity Assumption:** Given m_x ancestors X and m_y descendants Y in an XML dataset, we assume that all the elements Y are uniformly distributed under all their ancestors X. That is, if we randomly select m'_x out of m_x elements X, these X elements will contain m'_y Y descendants, such that $m'_x/m_x \approx m'_y/m_y$.

The Node Containment Uniformity Assumption can be applied when n_{i+1} and the target node n correspond to the X and Y in the assumption respectively. Thus, we get $S_{\vec{Q}}(n_{i+1})/S_Q(n_{i+1}) \approx S_{\vec{Q}}(n)/S_Q(n)$. Recall that $S_{\vec{Q}}(n_{i+1})/S_Q(n_{i+1}) \approx S_{\vec{Q}'}(n_{i+1})\ /S_{Q'}(n_{i+1})$ (Equation 5.3). This gives us $S_{\vec{Q}}(n)/S_Q(n) \approx S_{\vec{Q}'}(n_{i+1})/S_{Q'}(n_{i+1})$, and we have:

$$S_{\vec{Q}}(n) \approx S_Q(n) * S_{\vec{Q}'}(n_{i+1})/S_{Q'}(n_{i+1}) \tag{5.4}$$

Example 5.10: Consider again the query $\vec{Q_1}$ in Figure 5.5(a). Suppose the target node is now D. Then $S_{\vec{Q_1}}(D)$ is estimated as $S_{Q_1}(D) * S_{\vec{Q'_1}}(B)/S_{Q'_1}(B) = 1.3 * 2/2.6 = 1$, where the value of $S_{\vec{Q'_1}}(B) = 2$ is retrieved from the *path-order table* for B, and $S_{Q_1}(D)$ and $S_{Q'_1}(B)$ are estimated as 1.3 and 2.6 respectively. □

Case 2: Target Node in Trunk Part

When the target node n occurs in the trunk part q_1 of $\vec{Q}$, it is obvious that the selectivity $S_{\vec{Q}}(n)$ must not be larger than $S_Q(n)$, the upper bound of $S_{\vec{Q}}(n)$. In addition, we can further optimize the estimation with order information.

We observe that when the order axes of $\vec{Q}$ is imposed on the query Q, some elements n_{i1} (the first node of q_2) that do not satisfy the order axes will be eliminated from the query result sets. According to the Node Containment Uniformity Assumption, these eliminated n_{i1} elements are uniformly distributed under all elements n that satisfy the query Q (without order axes), and so are the remaining elements n_{i1}. Thus, we can deduce that the elimination of element n_{i1} as a result of imposing the order axis does not affect the selectivity $S_Q(n)$ if the number of remaining elements n_{i1}, i.e., $S_{\vec{Q}}(n_{i1})$, is greater than or equal to $S_Q(n)$. When $S_{\vec{Q}}(n_{i1})$ is less than $S_Q(n)$, each element n in $\vec{Q}$ has at most one descendant n_{i1}, and thus the value of $S_{\vec{Q}}(n)$ is estimated as $S_{\vec{Q}}(n_{i1})$. Similarly, we can optimize the upper bound estimation with $S_{\vec{Q}}(n_{i+1})$.

As a result, given a query $\vec{Q}$ where the target node n occurs in the trunk part, we have:

$$S_{\vec{Q}}(n) \approx min(\ S_Q(n), S_{\vec{Q}}(n_{i1}), S_{\vec{Q}}(n_{i+1})\ ) \tag{5.5}$$

5.4.2 Preceding/Following Axis

The techniques to estimate queries with a *preceding-sibling* (*following-sibling*) axis can be easily extended to process queries with a *preceding* (*following*) axis. The basic idea is that we can convert a query with a *preceding* (*following*) axis into a set of XPath expressions involving only *preceding-sibling* (*following-sibling*) axes according to the path ids of the node associated with a *preceding* (*following*) axis after path id join. Then the estimation result is given by the sum of the selectivity of the set of path expressions. The following example illustrates:

Example 5.11: Consider the query $//A[/C/foll::D]$ ($foll$ denotes *following* axis) with target node D issued on the XML data of Figure 5.1. The path id join associates nodes A, C and D with path id sets $\{p_6, p_7\}$, $\{p_2, p_3\}$ and $\{p_5\}$ respectively. We check the path id p_5 of node D. Since only the first bit of $p_5(1000)$ is 1, the path between A and D must be $A/B/D$ (look up *encoding table* with value 1). As a result, the given query can be converted to a query with a *following-sibling* axis: $//A[/C/folls::B/D]$. □

5.5 Data Structures

The *pathId-frequency table* and the *path-order table* are implemented by the *p-histogram* and *o-histogram* respectively. Since both histograms are constructed from the path ids of the element nodes in an XML document, we need a quick way to access these path ids. This task is achieved with the use of a binary tree to index path ids. This section describes the *path id binary tree*, *p-histogram* and *o-histogram*.

5.5.1 Path ID Binary Tree

We use a binary tree to index path ids. The structure of the binary tree is defined as follows:

1. The left and right edge in the binary tree represent the bit 0 and 1 respectively.

2. Each leaf node represents a path id, which is specified by the id (integer) associated with the node.

3. The bit sequence of the path id at each leaf node can be obtained by concatenating bits of all edges from the root node to this leaf node.

4. The id attached with an internal node is the largest path id in its left subtree. If the left subtree of an internal node is empty, this node is attached with an integer that is less than the least value in its right subtree.

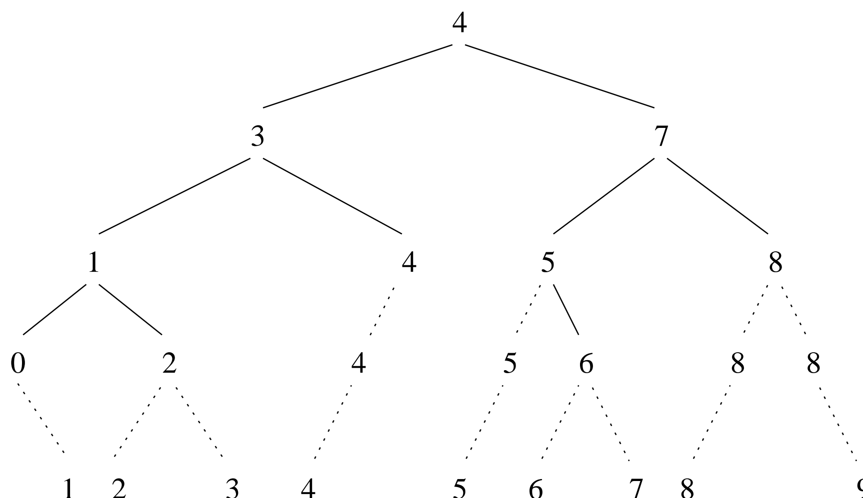


Figure 5.6: Path Id Binary Tree

Example 5.12: Figure 5.6 shows the binary tree of the path ids in Figure 5.1(c). The leftmost internal node is assigned the value 0 while the least path id value in its right subtree is 1. The leaf node with id 2 denotes the path id 0010 which is

obtained by concatenating the bits of all edges in the path from the root to the leaf. □

To find the bit sequence of a given path id, we can simply navigate down the binary tree starting from the root node. During this process, we compare the given path id with the ids of the internal nodes, and visit the left child if the given value is not greater than the id of the node, or the right child otherwise. After reaching the leaf node, the concatenation of the bits of all edges traversed is the bit sequence of the given path id.

The binary tree can be compacted without losing the path id information. The idea is that if a left (right) subtree of an internal node only contains left (right) edges, we remove this left (right) subtree and its incoming edge. This is because this left (right) subtree only represents a subsequence containing all 0 (1) bits.

Example 5.13: Consider Figure 5.6 again. All the dotted edges and the nodes under them can be safely removed from the binary tree. □

5.5.2 P-Histogram

To summarize pathId-frequency information, a *p-histogram* is built for each distinct element tag. Each bucket in a *p-histogram* contains a set of path ids and their average frequency value. To reduce the data skewness inside a bucket, we require that the frequency variance of each bucket is not larger than a given variance threshold v. Given a set of pathid-frequency pairs (p_i, f_i) for one element tag, the frequency variance v_b of a bucket is defined as follows:

$$v_b = \sqrt{\frac{(f_1 - avgf)^2 + ..(f_k - avgf)^2}{k}} \ ,$$

where f_i denotes the frequency of a path id p_i, and k is the number of path ids in the bucket, and $avgf = \sum f_i/k$.

Path_id–Frequency	(p2, 2) (p3, 2) (p1, 5) (p5, 7)		
P–Histogram1 v = 0	(fre = 2 p2, p3) v = 0	(fre = 5 p1) v = 0	(fre = 7 p5) v = 0
P–Histogram2 v = 1	(fre = 2 p2, p3) v = 0	(fre = 6 p1, p5) v = 1	

Figure 5.7: P-Histogram

Example 5.14: Figure 5.7 shows two *p-histograms* built on the same pathid-frequency list with different given variance values, 0 and 1. Note that the variance with value 0 indicates the intra-bucket frequency must represent the correct frequency values of the corresponding path ids. □

Construction

Building a perfect variance optimal histogram (V-Optimal) is beyond the scope of this paper. Here, we give a simple yet efficient heuristic algorithm to build a *p-histogram* (see Algorithm 11).

The *p-histogram* construction algorithm takes the pathId-frequency list for an element e and a variance threshold v as input. This pathId-frequency list is first sorted according to frequency values. Next, we scan the list and find the longest sublist such that its frequency variance is not greater than the given threshold v. The data in the longest sublist detected is then used to build a bucket. This detect-and-

Algorithm 11 P-Histogram Construction

Input: PathId-frequency list for element e and variance threshold v
Output: P-histogram for element e

1. Sort the pathId-frequency list according to the frequency values.
2. Create a bucket b. Scan and add the path ids into b until the intra-bucket variance is larger than the given v.
3. Repeat step 2 until scanning of the pathId-frequency list is complete.

build procedure is repeated until we have completely scanned the pathId-frequency list.

Example 5.15: Figure 5.7 shows the two p-histograms constructed using Algorithm 11 with variance values 0 and 1 respectively. □

Maintenance

When the underlying pathId-frequency data is modified, two simple steps can be utilized to update the associated *p-histogram*.

Step 1: Merge. If updates occur outside any bucket of the *p-histogram*, we merge the updated data with the adjacent buckets. If the updated data happens to be in some buckets, this step is ignored.

Step2: Split. For any bucket whose variance is greater than the defined variance bound value, we call Algorithm 11 by passing the intra bucket data. This step produces several new buckets which satisfy the intra-bucket variance requirement.

5.5.3 O-Histogram

The *o-histogram* summarizes path-order information. We observe that the *path-order table* is very sparse since the frequencies in the majority of the cells are 0 (or empty). We only need to store the cells with non-zero values.

A bucket in the *o-histogram* has the format of (*x.start, y.start, x.end, y.end, frequency*) where the first four variables of the bucket describe a bounding box in the *path-order table* and the variable *frequency* denotes the average frequency value of all cells in this box. Similar to the p-histogram, the *o-histogram* also uses frequency variance to reduce intra-bucket data skewness.

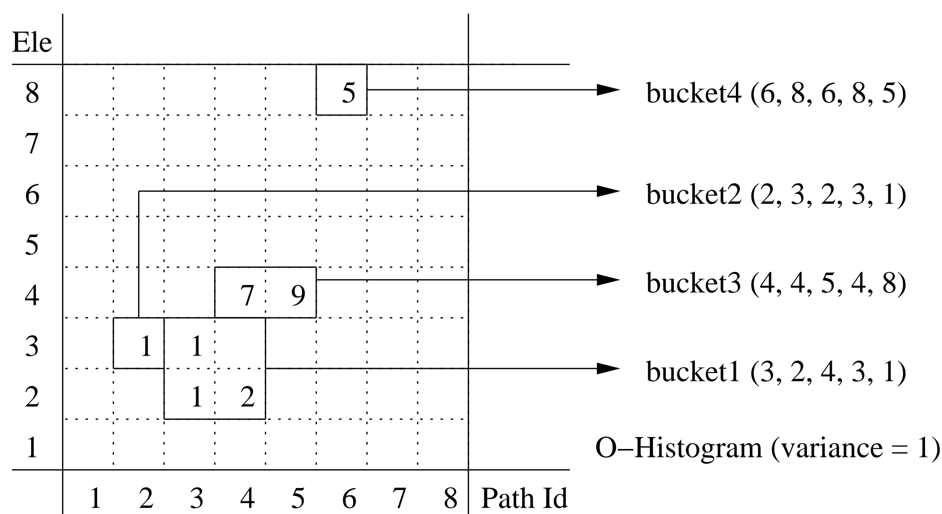


Figure 5.8: O-Histogram

Example 5.16: Figure 5.8 shows an *o-histogram* that is built on the given *path-order table* with variance 1. This *o-histogram* has four buckets and they cover all the non-empty cells in the table. □

Construction

Algorithm 12 shows the details of constructing the *o-histogram.* First, we sort the given *path-order table* according to the alphabetical order of the element tags and the path id order generated in the *p-histogram.* The sorted element tags and path ids are then encoded using integer numbers. The purpose of this step is that given a bounding box (*x.start, y.start, x.end, y.end*), we can find the corresponding element tags and path ids.

Algorithm 12 O-Histogram Construction

Input: Path-order table for element e and variance threshold v
Output: O-histogram for element e

1. Sort path-order table:
   - Element tags: alphabetical order.
   - Path ids: Path ids order in p-histogram for e.
2. Scan non-empty cells in path-order table to get a bounding box (bucket) as follows:
   - Extend the current cell to a row of cells.
   - Extend this row to a box of cells.
   - In each of the extension steps, the intra-bucket variance is not larger than the given v.
3. Repeat step 2 until we finish scanning all non-empty cells.

Next, all the non-empty cells in the *path-order table* are scanned row-wise. For each non-empty cell, we extend it to a possible maximal box such that the frequency variance of the cells within the box is less than or equal to the given variance threshold.

The detection of the maximal box is performed in two steps. First, we extend the current cell to a row of cells. This extension stops if we encounter an empty cell, or the next cell is inside some other well-built bucket. Second, this row of cells is extended to a box by adding the rows above this row to the bucket until an empty-row (all cells are empty) is reached. In each step of the extension, we must guarantee that the variance in the box (or in just one row) is not larger than the given variance value.

Example 5.17: The *o-histogram* shown in Figure 5.8 is built by using Algorithm 12 with the variance value 1. After sorting the element tags and path ids, we check

the cell (3,2) and extend it to a row (3,2,4,2) and box (3,2,4,3) respectively. Next, we scan the rest of the non-empty cells. We ignore the cell (4,2) since it is already in the existing bucket. After that, bucket 2 is built on the single cell (2,3), etc. □

Maintenance

As with the *p-histogram*, we have two steps *merge* and *split* to update the *o-histogram*. We merge the updated data and the adjacent buckets into some bigger bounding boxes, and then rebuild buckets on these new boxes. Note that if there are no buckets adjacent to the updated data (surrounded by empty rows and columns), we can directly carry out the split step on the data to build new buckets.

5.6 Experiments

We carry out experiments to evaluate the performance of the proposed techniques in terms of memory space requirement, summary construction time and estimation accuracy.

Datasets

We use both real-world and synthetic datasets. Table 5.1 shows the characteristics of the datasets: Shakespears's Play (SSPlays) [1], DBLP [4] and XMark [2]. Attributes are omitted for simplicity.

Dataset	Size	♯(Distint Eles)	♯(Eles)
SSPlays	7.5 MB	21	179,690
DBLP	65.2 MB	31	1,711,542
XMark	20.4 MB	74	319,815

Table 5.1: Characteristics of Datasets

Query Workload

We first generate 4000 simple and 4000 branch queries without order axes for each dataset. The simple queries are generated by randomly selecting the subsequences of the root-to-leaf paths from the *encoding table*. The branch queries are produced by merging any two of these subsequences if they have common nodes. The query sizes (number of nodes) vary from three to 12. Duplicate queries and negative queries are removed to obtain a reasonable average relative error.

Datasets	Without Order			With Order
	Simple	Branch	Total	
SSPlays	188	2328	2516	1168
DBLP	202	1013	1215	646
XMark	1358	2686	4044	1654

Table 5.2: Query Workload

Next, we generate queries with order axes by fixing the order between the sibling nodes of the generated branch queries and then eliminating negative ones from them. Table 5.2 shows the number of queries obtained. Note that for simple and branch queries without order axes, we randomly select their target nodes. For queries with order axes, we randomly select one node in the trunk part as well as in the branch part.

5.6.1 Memory Space Requirement

We first evaluate the space requirement of the *encoding table* and *path id binary tree*. Table 5.3 shows the results.

We observe that the sizes of *encoding table*s are very small for all datasets. To show the space savings of the binary tree, we also give the size of the *path id table*. The real-world datasets (SSPlays and DBLP) require very limited space even if we do not use binary tree. This is due to their regular structures, and hence, fewer number of distinct paths (40 and 87 respectively). In contrast, the binary tree

is able to save about 78% of the space requirement for the XMark dataset. This is because XMark has a large number of distinct paths, leading to long path ids. Many large subtrees containing only left (or right) edges can be trimmed off from the binary tree.

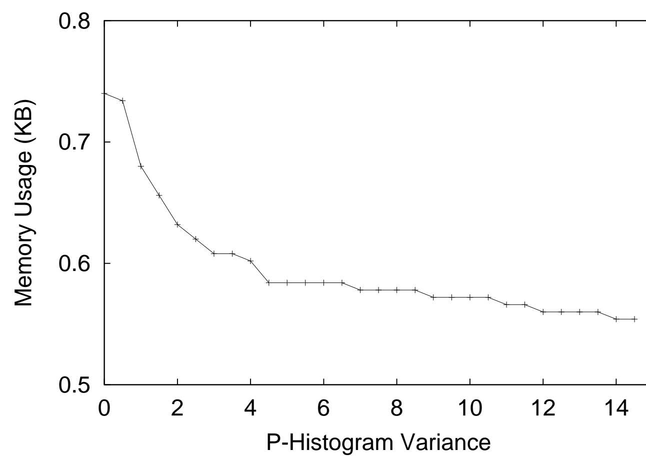
Dataset	♯(Dist Paths)	Pid Size (Byte)	♯(Dist Pid)
SSPlays	40	5	115
DBLP	87	11	327
XMark	344	43	6811

Dataset	EncTab (KB)	PidTab (KB)	Pid Bin-Tree (KB)
SSPlays	0.24	0.92	0.93
DBLP	0.39	3.60	2.97
XMark	2.90	299.7	67.3

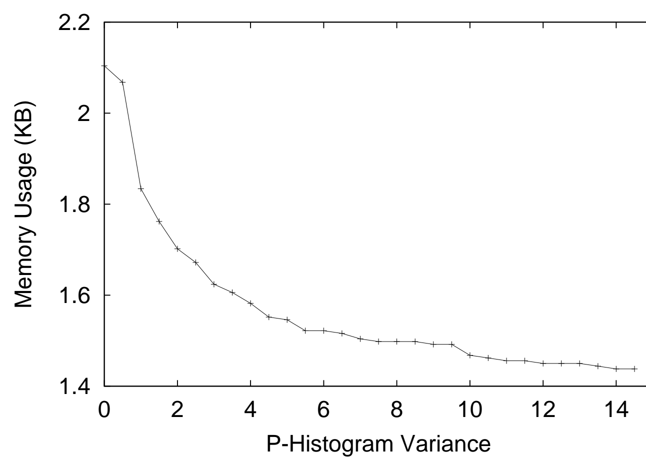
Table 5.3: Space Requirement of Encoding Table and Path Id Binary Tree

Next, we examine the memory requirement of the *p-histogram* and *o-histogram*. Figure 5.9 shows the memory usage of the *p-histogram* at varying intra-bucket variance values. We see that all the datasets have similar curves. The *p-histogram* memory usage decreases when the variance value is varied from 0 to 4. The dataset XMark needs more memory space compared to the other datasets since it has more element tags and distinct path ids.

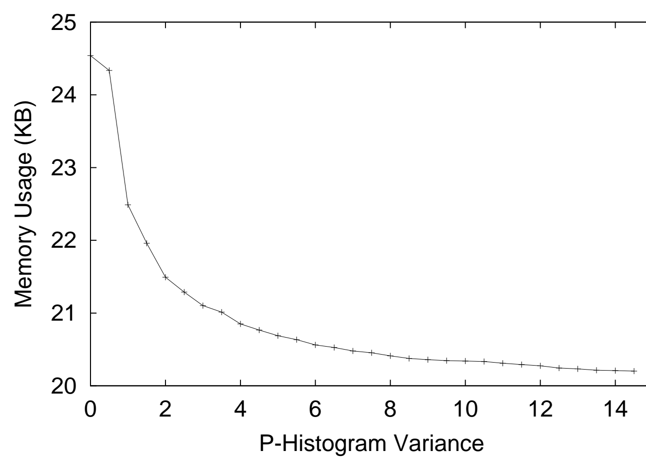
Figure 5.10 shows that *o-histogram* memory usage for all three datasets decreases as *o-histogram* intra-bucket variance grows. Comparing memory curves, the *p-histogram* and the *o-histogram* require nearly the same memory space for the SSPlays and XMark datasets while we see a sharp increase of memory usage from *p-histogram* to *o-histogram* for the DBLP dataset. This is because the data distribution in DBLP is shallower and wider than in the other datasets. As a result, the large number of sibling nodes in DBLP generates more order information to be stored.



(a) SSPlays

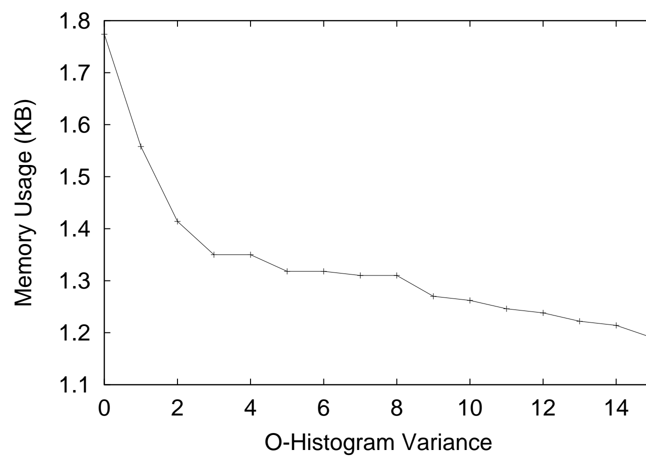


(b) DBLP

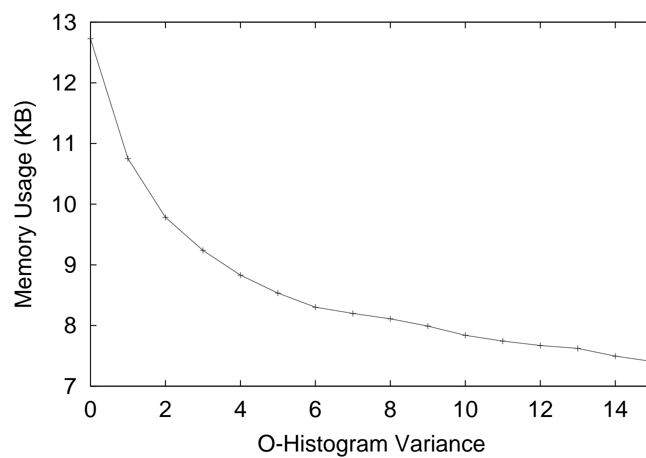


(c) XMark

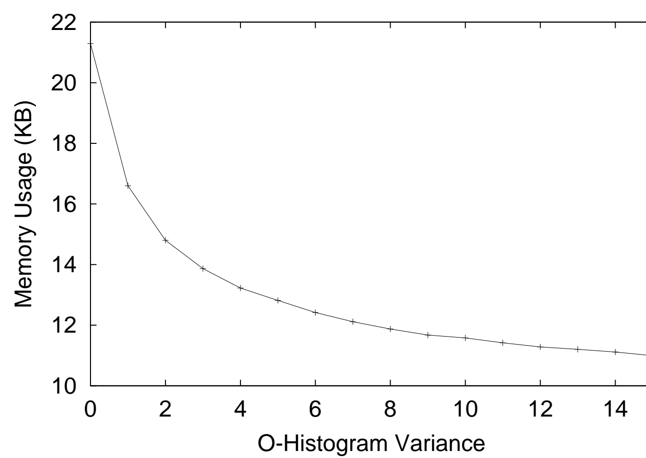
Figure 5.9: P-Histogram Memory Usage



(a) SSPlays



(b) DBLP



(c) XMark

Figure 5.10: O-Histogram Memory Usage

5.6.2 Summary Construction Time

In this experiment, we evaluate the cost of constructing summary information for selectivity estimation. Table 5.4 shows the construction time of the proposed solution and *XSketch* [64] for queries without order axes. In our proposed solution, the XMark dataset requires the longest time for the path information to be summarized since it has the most number of distinct root-to-leaf paths. We observe that the *p-histogram* construction time is almost negligible for all three datasets. This is because our algorithm scans the path-frequency information only once to build the *p-histogram* (Algorithm 11).

Proposed Path-Based Solution			
Dataset	Collecting Path Time	P-Histo Size	P-Histo Construct Time
SSPlays	1.6 S	0.55 $\sim$ 0.75 KB	<0.001 S
DBLP	78.4 S	1.4 $\sim$ 2.1 KB	<0.001 S
XMark	246.2 S	20.4 $\sim$ 24.6 KB	<0.001 S

XSketch			
Dataset	Collecting Summary Time	Statistics Size	Statistics Construct Time
SSPlays	32.3 S	1.6 $\sim$ 2 KB	2 $\sim$ 3 S
DBLP	390.7 S	4.8 $\sim$ 5.8 KB	19 $\sim$ 30 S
XMark	197.7 S	90 $\sim$ 95 KB	> 1 Week

Table 5.4: Summary Construction Time for Queries without Order Axes

In contrast, [64] shows that building an optimal *XSketch* model is an *NP*-hard problem as introduced in Chapter 4. Hence, it utilizes a greedy refinement strategy to incrementally add statistics on the existing summary information. As a result, the construction time grows quickly when the statistics size increases. In Table 5.4, the construction time of the XMark dataset with the statistics size of 90-95 KB even reaches more than one week, which is unacceptable in practice. Note that we ensure the summary size of *XSketch* is approximately the same as the total

memory size of our solution, that is, the summary of the *encoding table*, the *path id binary tree* and the *p-histogram*.

Dataset	Collecting Order Time	O-Histo Size	O-Histo Construct Time
SSPlays	2.2 S	1.2 $\sim$ 1.8 KB	0.002 $\sim$ 0.003 S
DBLP	4574.8 S	7.4 $\sim$ 12.7 KB	0.02 $\sim$ 0.03 S
XMark	2347.2 S	11 $\sim$ 21.3 KB	1.2 $\sim$ 2.1 S

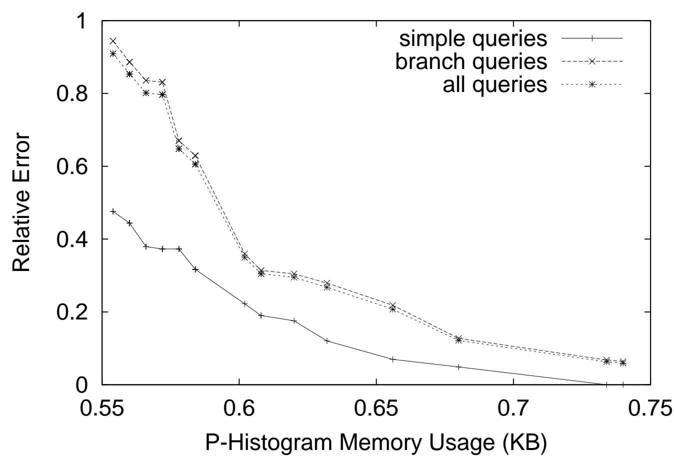
Table 5.5: Summary Construction Time for Order Data

Table 5.5 presents the construction time for order data. Compared with Table 5.4, the order data collection consumes more time for all three datasets due to the huge amount of order information that needs to be captured. Similar to the *p-histogram*, the *o-histogram* building algorithm remains efficient because of the simple one-scan construction method.

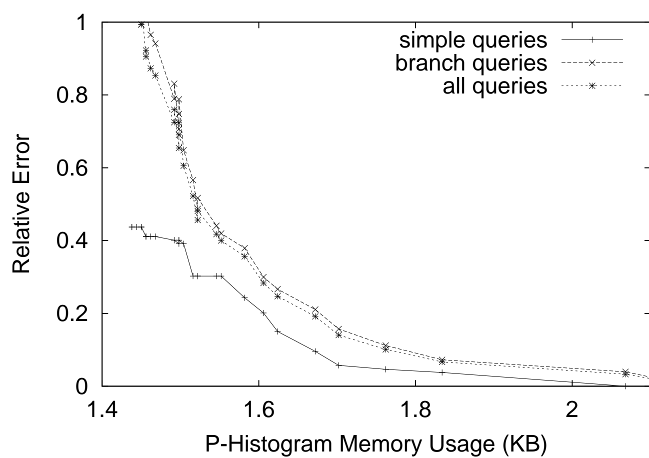
5.6.3 Estimation Accuracy of Queries without Order Axes

In this section, we evaluate the accuracy of the proposed solution on XPath queries without order axes, and we compare our approach with $XSketch$ [64].

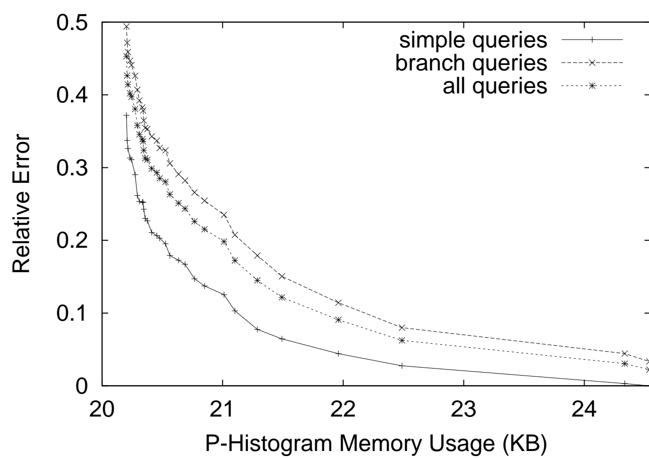
Figure 5.11 shows the estimation accuracy for queries without order axes. The memory usage in the x-axis corresponds to the memory usage in the y-axis of the *p-histogram* in Figure 5.9. The last data points, which correspond to *p-histogram.variance* = 0, show the correct selectivity obtained for simple queries. We observe that the error of branch queries is very low (less than 7% for all datasets) when the *p-histogram* variance is zero. Simple queries have a lower estimation error than branch queries do. The larger estimation error of the branch queries arises from the estimated data (bucket frequency). Further, branch queries may not satisfy the Node Independence Assumption.



(a) SSPlays

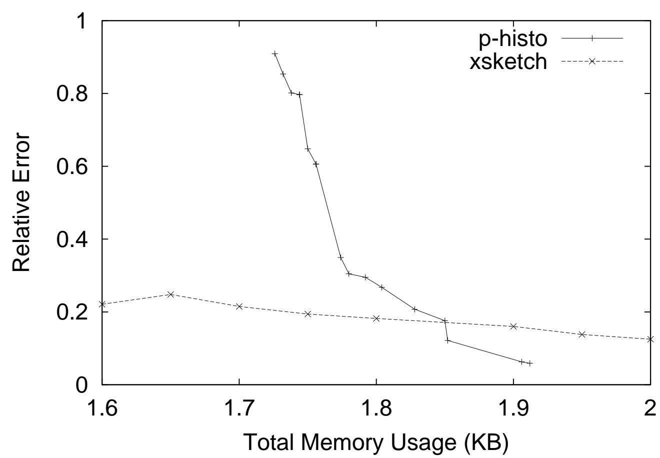


(b) DBLP

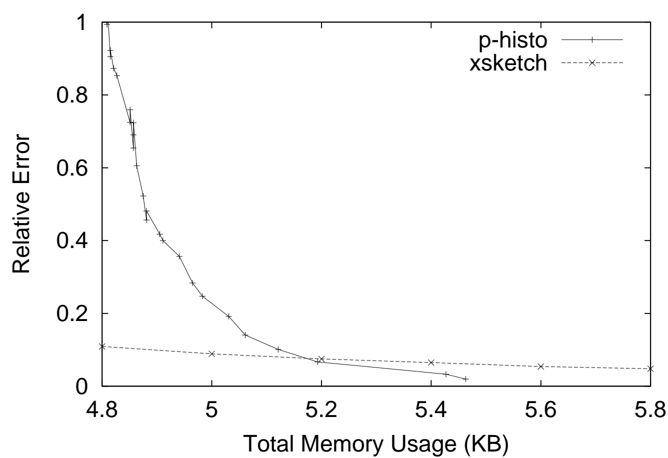


(c) XMark

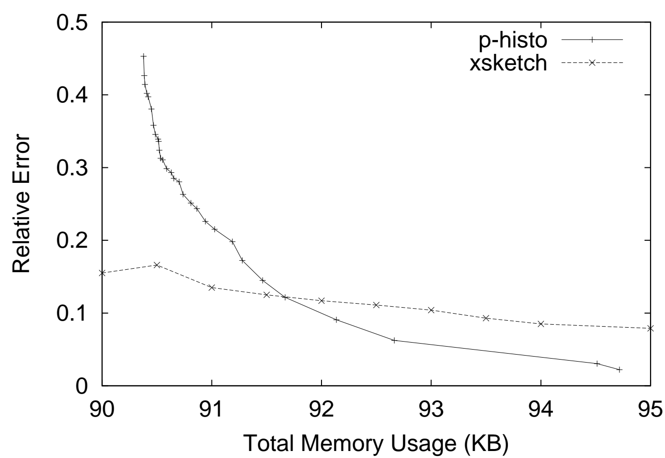
Figure 5.11: Estimation Error of Queries without Order Axes



(a) SSPlays



(b) DBLP



(c) XMark

Figure 5.12: P-Histogram vs. XSketch

Comparison with XSketch

We also compare the proposed estimation method with $XSketch$ [64] for queries without order axes. Figure 5.12 shows the total memory usage of our approach (*encoding table*, *path id binary tree* and *p-histogram*). Note that the curves for our method are shorter than that for $XSketch$. This is because maximal memory usage occurs when the *p-histogram* variance is 0 (corresponding to the last data points of the curves) for our method. We observe that if sufficient memory space (corresponding to *p-histogram* variance from 0-2) is available, our method outperforms $XSketch$. $XSketch$ shows more accurate results with low memory usage. This is because our solution requires a minimum space to store the *encoding table* and *path id binary tree*, and additional memory space (*p-histogram*) increases estimation accuracy, leading to a significant decrease in estimation error. In contrast, $XSketch$ captures the summarized structure of XML data, which does entail the minimum memory requirement that our method does. Hence, it can work under low memory space and has a relatively flat estimation error curve.

Comparison with NR-NF Solution

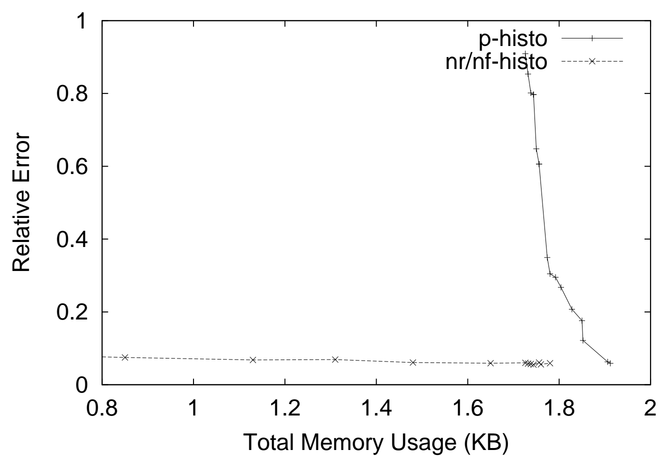
This part compares the proposed path-based estimation method with the NR-NF approach which uses a fixed VF value ($VF = 2$) and various UF values ($100 < UF < 1500$) to control histogram construction. Figure 5.12 shows the results. We observe that for NR-NF approach, memory usage and estimation accuracy hit their limits when the UF value is greater than 1000. That is, the NR-NF approach cannot further detect data skewness when the granularity of a bucket assumes some small value (the bigger UF is, the smaller granularity is). As a result, the NR-NF solution consumes less memory space than the *p-histogram* solution does as shown in the graph. In contrast, the *p-histogram* method first col-

lects detailed statistical information (pathId-frequency table) and then compresses it by building histograms. This is totally different from the *NR-NF* solution, which extracts highly summarized data and then adds complimentary data skewness information. Therefore, the *p-histogram* method requires more memory space and can finally reach more accurate estimation results with adequate space. We also observe that for regular real-world datasets, the *NR-NF* approach is sufficient to provide accurate selectivity estimation while the *p-histogram* can achieve more satisfying results for synthetic XML data.

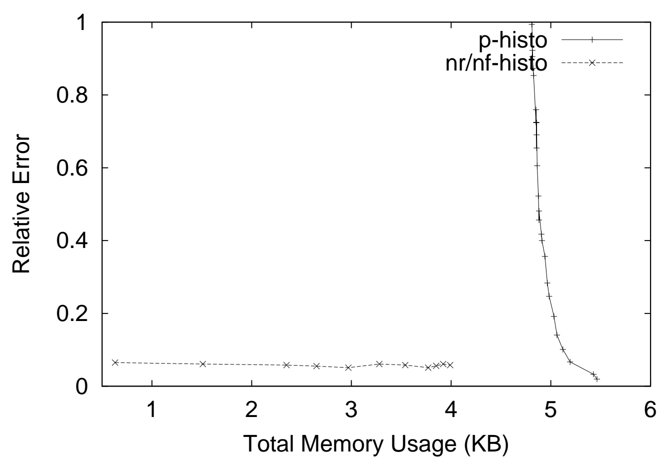
5.6.4 Estimation Accuracy of Queries with Order Axes

Next, we examine estimation accuracy for queries with order axes. Figure 5.14 shows the average relative error (target nodes in branch parts) when the memory usage of the *o-histogram*s varies. We set the variance of the *p-histogram* at 0, 1, 5 and 10, and plot four curves in each graph. We see that when the exact frequency values are stored in the *p-histogram* (*p-histogram.variance=0*), the relative errors for the three datasets are smaller than 10% at *o-histogram* variance 2 (the *o-histogram* memory usages are 1.4KB, 9.8KB and 14.8KB respectively for the three datasets), and the error rate can be further reduced to less than 6% when *o-histogram* variances are 0 (the memory usages are 1.8KB, 12.7KB and 21.3KB respectively).

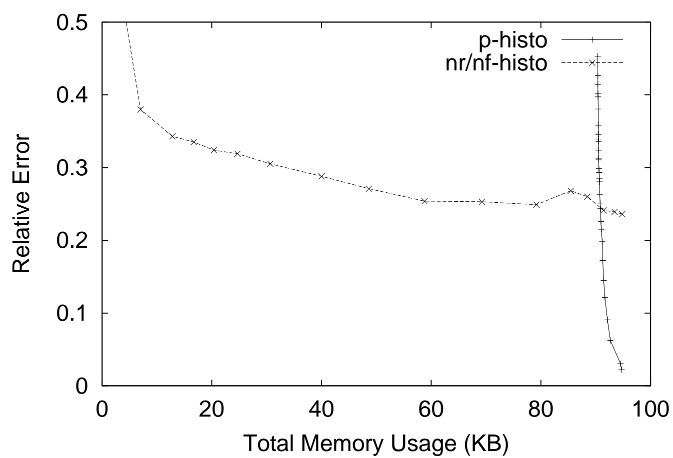
The accuracy curves for the SSPlays and XMark datasets are relatively flat at high *p-histogram* variance. This indicates that if the pathId-frequency information is not accurate (high *p-histogram* variance values), we cannot improve the estimation accuracy by setting smaller *o-histogram* variance (for more accurate order information). The curves for the DBLP dataset are very flat, indicating that this dataset is not sensitive to *o-histogram* variance in all values of *p-histogram* vari-



(a) SSPlays

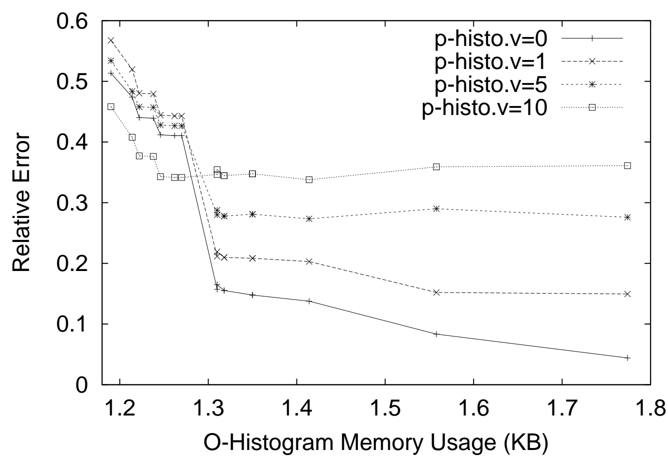


(b) DBLP

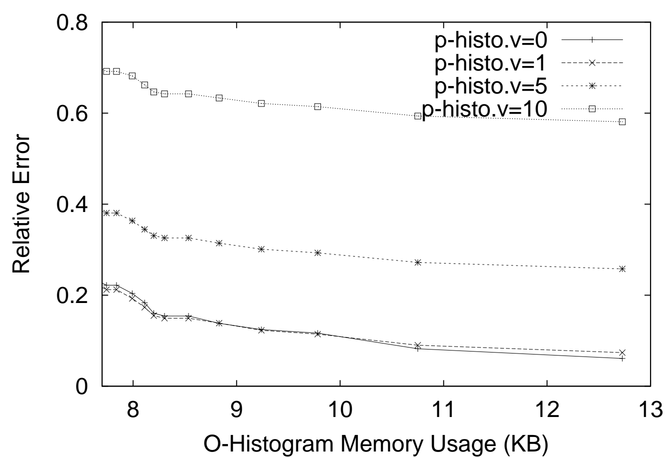


(c) XMark

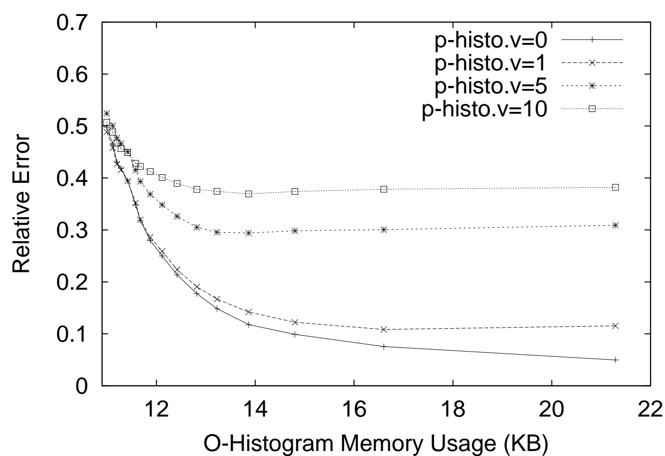
Figure 5.13: P-Histogram vs. NR-NF Histogram



(a) SSPlays



(b) DBLP



(c) XMark

Figure 5.14: Estimation Error of Queries with Order Axes (Branch Part)

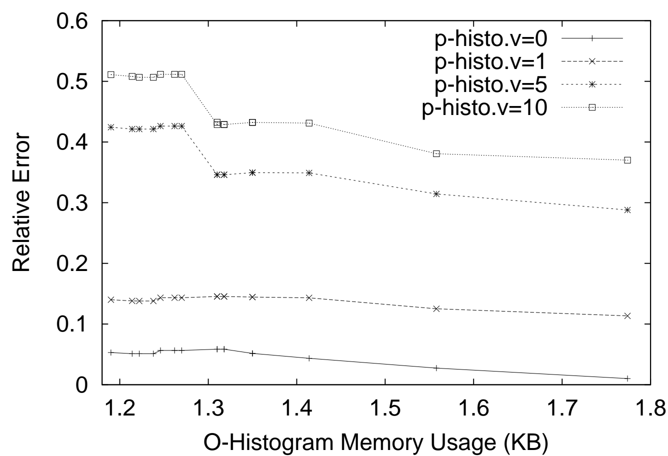
ance. This is because more memory space is required to store order information for the DBLP dataset compared to path information.

Figure 5.15 shows the estimation accuracy results when the target nodes occur in the trunk parts of the queries. We observe that the estimation is reasonably accurate at low *p-histogram* variances even if we set a high *o-histogram* variance (low *o-histogram* memory usage). Compared with Figure 5.14, we can achieve lower estimation error for the SSPlays and XMark datasets at low *p-histogram* variance values and low *o-histogram* memory usage (see Figure 5.15). This is because we use Equation 5.5 to estimate the selectivity which is the smallest value of the results of two order-based and one non-order-based queries. With the low *p-histogram* variance value, we can obtain accurate results for queries without order axes, which compensates for the loss of detailed order information.

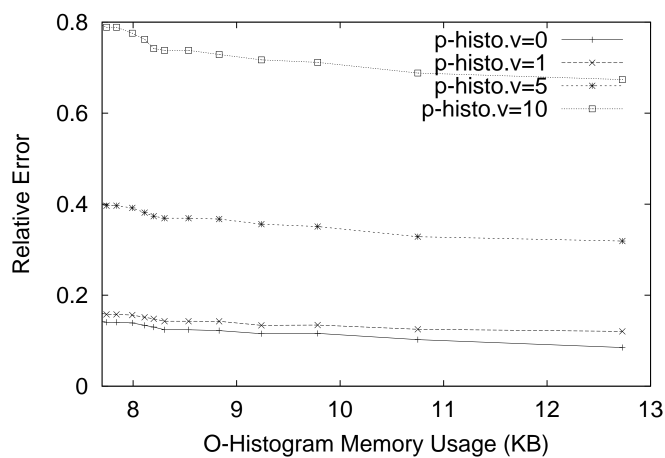
Overall, the experimental results demonstrate the effectiveness of the proposed techniques which yield low estimation error while requiring a very limited amount of memory. We also show that the proposed techniques typically perform well when *p-histogram* variance is set at 0-2 and *o-histogram* variance is set at 0-4.

5.7 Conclusion

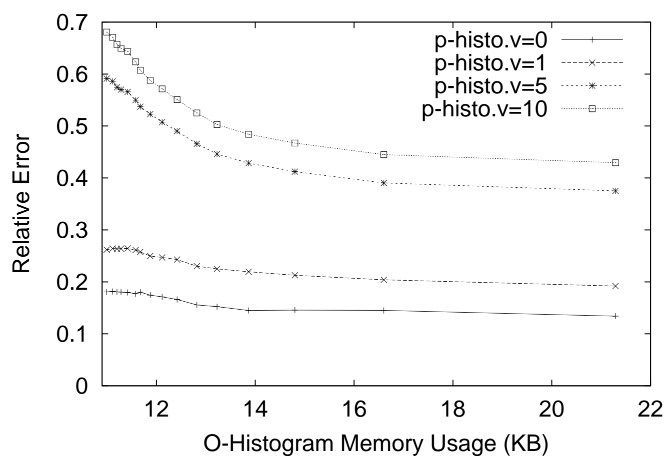
In this chapter, we design a uniform framework to estimate the selectivity of both XPath expressions with and without order axes. We capture the path information where the element occurs and utilize a join based method to estimate queries without order axes. The order information of each element tag is summarized with the use of a *path-order table*. We design two novel histograms, namely, the *p-histogram* and the *o-histogram*, to summarize the path information and order information respectively. We carry out extensive experiments to evaluate the proposed method.



(a) SSPlays



(b) DBLP



(c) XMark

Figure 5.15: Estimation Error of Queries with Order Axes (Trunk Part)

For queries without order axes, our proposed solution shows it has more accurate estimation results with sufficient available memory space and needs much more less summary construction time compared with existing approach. Moreover, to the best of our knowledge, this is the first work that provides estimation solution for queries with order axes. Experimental evaluation on both real-world and synthetic datasets clearly demonstrates the effectiveness of our proposed approach.

CHAPTER 6

Conclusion

With the development of the Internet, the World Wide Web is rapidly embracing XML as a new standard for data representation and exchange on the Web. As a result, XML query optimization becomes one of the most active and exciting research areas in the database community.

In XML query optimization systems, all query processing solutions rely on a pattern-specification language, such as XPath [10] or XQuery [7], that allows path navigation and branching through the XML data in order to reach the desired data elements. Building effective index structure to speed up XML query evaluation naturally becomes the core of query processing approaches. On the other hand, optimizing such queries depends crucially on the existence of concise structures that enable accurate compile-time selectivity estimates for complex path expressions over XML data. This thesis has described novel approaches for these two important components in the XML query optimization system: XML query processing and selectivity estimation.

6.1 Summary of Main Findings

This section summarizes the main findings of the thesis. We discuss the contributions to query processing and selectivity estimation respectively.

6.1.1 XML Query Processing

Among the existing techniques to evaluate XML queries, structural join is the defacto standard in that this method fundamentally solves the problem of specifying the containment relationship between nodes by utilizing join operation. Based on the structural join method, numerous approaches to processing XML queries have been developed, including the stack-tree [17], the B^+-tree [57], the R-tree [57], the XB-tree [19], etc. However, all these approaches fail to produce a satisfying performance, especially for Internet-scale XML data. This motivates the proposal of a more efficient solution for XML query processing.

Based on the observation that the paths in an XML document play crucial roles in connecting elements, we have designed a novel XML labeling scheme and a corresponding path join algorithm. The path-based labeling scheme associates each element in an XML document with a pair (path id, node id) while the path join algorithm eliminates irrelevant path types which do not contribute to the final result sets.

We have performed extensive experiments to check the query performance of the proposed path-based solution. The comparison of the proposed approach against the state-of-the-art access method, the XB-tree based holistic join TwigStack [19] and the path index approach BLAS [25] demonstrate with certainty that our proposed path-based solution significantly outperforms the other two methods due to the ability of path join to efficiently eliminate unnecessary path types. This can be

explained as follows:

First, path list size is much smaller than node list size. This is expected since path ids capture summarized path information while node ids specify the detail of each node. The small size of path lists guarantees the low cost of path join.

Second, path ids essentially capture the information of paths in which the element nodes occur. As discussed in the algorithm, path join generates sets of elements that are as minimal as possible. In the case of simple queries, the path types associated with element nodes can be reduced to an exact path id set. For branch queries, path id join generates candidate element sets that are smaller than those generated by BLAS [25].

6.1.2 XML Query Selectivity Estimation

There is a long stream of literature on XML query selectivity estimation. Early work in the area supports a limited class of XML queries, simple queries. Examples of such work include the Markov-based solutions [15, 62, 58], the path-tree [15], and *position histogram* [83] etc. The more recent work has focused on the selectivity estimation for branch queries (twig queries). Examples include the solution in [26] and the *XSketch* family [64, 65, 66]. However, *XSketch* family suffers the problem of expensive construction time due to the complicated underlying data structure employed.

In this thesis, we have proposed two approaches of XML query selectivity estimation. The first solution extracts two pieces of summarized information: *Node Ratio* and *Node Factor* from distinct parent-child paths. Given an XML query, we have designed an effective and efficient method to aggregate the summarized information based on the proper Basic Path Independence Assumption to calculate query selectivity. The experimental results show that this solution

requires very little memory space, but yet provide accurate estimation results for regularly distributed real-world XML data. For skewed XML data, histograms are built to effectively capture the distribution of the underlying data.

The second estimation approach utilizes the path-based labeling scheme proposed for query processing to collect the statistical information of XML nodes. Compared with the first solution, this approach consumes more memory space but provides more accurate estimation results. In addition, this solution, to the best of our knowledge, is the first work to address the problem of estimating XML queries with order-based axes. We have designed a succinct data structure, o-histogram to capture the huge order information existing in XML data. Extensive experiments clearly demonstrate the efficiency of the proposed solution.

6.2 Future Work

While this thesis has presented efficient approaches to XML query processing and selectivity estimation, a number of issues need to be further investigated:

- First, three approaches proposed in the thesis have focused on tree structured XML data, and further study can be conducted to extend these solutions to handle graph-based XML models. Since the graph models of XML data contain more information (ID references) than tree models do, we expect the proposed path-based labeling scheme to be revised to collect ID references information between XML element nodes. The path information collected can help in the processing of XML queries with ID references.

- Second, both query selectivity estimators proposed in the thesis have focused on XML queries without value predicates. This is because both solutions do not capture the distribution information of text values. To overcome this

problem, text value distribution information should be properly summarized and combined into the selectivity estimation methods.

- Third, one open problem of selectivity estimation is how XML queries with aggregation functions should be handled. For example, we may want to find all the professors in the university who have more than 10 publications in the past year. This query contains the aggregation function "count()", and all existing XML estimation methods cannot process this case. To handle this class of queries, we should capture more detailed distribution information of XML elements.

BIBLIOGRAPHY

[1] http://www.ibiblio.org/xml/examples/shakespeare.

[2] http://monetdb.cwi.nl/.

[3] http://www.imdb.com.

[4] http://www.informatik.uni-trier.de/~ley/db/.

[5] http://www.yahoo.com.

[6] http://www.google.com.

[7] An XML Query Language (XQuery). http://www.w3.org/XML/Query.

[8] Extensible Markup Language (XML). http://www.w3.org/XML/.

[9] XML Document Type Definition (DTD). http://www.w3.org/TR/REC-xml/.

[10] XML Path Language. http://www.w3.org/TR/xpath.

[11] XML Query Use Cases. http://www.w3.org/TR/xquery-use-cases/.

[12] XML Schema. http://www.w3.org/XML/Schema.

[13] S. Abiteboul. Querying Semi-Structured Data. In *Proceedings of Database Theory, 6th International Conference, Delphi, Greece*, 1997.

[14] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Int. J. on Digital Libraries 1*, 1997.

[15] A. Aboulnaga, A. R. Alameldeen, and J. F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy*, 2001.

[16] S. Al-Khalifa and H. V. Jagadish. Multi-Level Operator Combination in XML Query Processing. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA*, 2002.

[17] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA*, 2002.

[18] P. Bohannon, J. Freire, P. Roy, and J.Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA*, 2002.

[19] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, 2002.

[20] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *Proceedings of the 3rd International Workshop on the Web and Databases, Dallas, Texas, USA (Informal proceedings)*, 2000.

[21] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt*, 2000.

[22] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. In *Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA*, 2004.

[23] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On Random Sampling over Joins. In *Proceedings ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA*, 1999.

[24] T. Chen, J. Lu, and T. W. Ling. On Boosting Holism in XML Twig Pattern Matching Using Structural Indexing Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA*, 2005.

[25] Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France*, 2004.

[26] Z. Chen, H. V. Jagadish, F. Korn, and N. Koudas. Counting Twig Matches in a Tree. In *Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany*, 2001.

[27] Z. Chen, F. Korn, N. Koudas, and S. Muthukrishnan. Selectivity Estimation For Boolean Queries. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Dallas, Texas, USA*, 2000.

[28] S-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[29] B. Choi, M. Mahoui, and D. Wood. On the Optimality of Holistic Algorithms for Twig Queries. In *Database and Expert Systems Applications, 14th International Conference, Prague, Czech Republic*, 2003.

[30] C. Chung, J. Min, and K. Shim. APEX: An Adaptive Path Index for XML Data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, 2002.

[31] E. Cohen, H. Kaplan, and T. Milo. Labelling Dynamic XML Tree. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Madison, Wisconsin, USA*, 2002.

[32] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy*, 2001.

[33] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proceedings ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA*, 1999.

[34] M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. *Computer Networks 33(1-6): 723-745 (2000).*

[35] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with eXtended Access Support Relations. In *Proceedings of the 3rd International Workshop on the Web and Databases, Dallas, Texas, USA*, 2000.

[36] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Eng. Bull. 22(3): 27-34 (1999).*

[37] J. Freire, J. R. Haritsa, M. Ramanath, R. Prasan, and J. Simeon. StatiX: Making XML Count. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, 2002.

[38] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases, Athens, Greece*, 1997.

[39] T. Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, 2002.

[40] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. Multi-Dimensional Substring Selectivity Estimation. In *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK*, 1999.

[41] H. V. Jagadish, O. Kapitskaia, R. T. Ng, and D. Srivastava. One-dimensional and Multi-dimensional Substring Selectivity Estimation. In *VLDB Journal (2000) 9*, 2000.

[42] H. V. Jagadish, R. T. Ng, and D. Srivastava. Substring Selectivity Estimation. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, 1999.

[43] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of the 19th International Conference on Data Engineering, Bangalore, India*, 2003.

[44] H. Jiang, W. Wang, and H. Lu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of 29th International Conference on Very Large Data Bases, Berlin, Germany*, 2003.

[45] E. Jiao, T. W. Ling, and C. Y. Chan. PathStack¬: A Holistic Path Join Algorithm for Path Query with Not-Predicates on XML Data. In *Database Systems for Advanced Applications, 10th International Conference, DASFAA, Beijing, China*, 2005.

[46] H. Kaplan, T. Milo, and R. Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, USA*, 2002.

[47] 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, Madison, Wisconsin*, 2002.

[48] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for Structure Indexes. In *Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[49] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA*, 2002.

[50] D. D. Kha, M. Yoshikawa, and S. Uemura. An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th International Conference on Data Engineering, Heidelberg, Germany*, 2001.

[51] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating Alphanumeric Selectivity in the Presence of Wildcards. In *Proceedings of the 1996 ACM SIGMOD*

International Conference on Management of Data, Montreal, Quebec, Canada, 1996.

[52] M. L. Lee, H. Li, W. Hsu, and B. C. Ooi. A Statistical Approach for XML Query Size Estimation. In *International Workshop on Database Technologies for Handling XML information on the Web, In conjunction with EDBT*, 2004.

[53] H. Li, M. L. Lee, and W. Hsu. A Histogram-Based Selectivity Estimator for Skewed XML Data. In *Database and Expert Systems Applications, 16th International Conference, Copenhagen, Denmark*, 2005.

[54] H. Li, M. L. Lee, and W. Hsu. A Path-Based Labeling Scheme for Efficient Structural Join. In *Third International XML Database Symposium, In Conjunction with VLDB, Trondheim, Norway*, 2005.

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

[56] H. Li, M. L. Lee, W. Hsu, and G. Cong. An Estimation System for XPath Expressions. In *Proceedings of the 22nd International Conference on Data Engineering, Atlanta, Georgia, USA*, 2006.

[57] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy*, 2001.

[58] L. Lim, M. Wang, S. Padmanabhan, J. S. Vitter, and R. Parr. XPathLearner: An On-Line Self-Tuning Markov Histogram for XML Path Selectivity Estimation. In *Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[59] 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).*

[60] J. Lu, T. Chen, and T. W. Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA*, 2004.

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

[62] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK*, 1999.

[63] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of Database Theory, 7th International Conference, Jerusalem, Israel*, 1999.

[64] N. Polyzotis and M. Garofalakis. Statistical Synopses for Graph-Structured XML Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, 2002.

[65] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity Estimation for XML Twigs. In *Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA*, 2004.

[66] N. Polyzotis and M. N. Garofalakis. Structure and Value Synopses for XML Data Graphs. In *Proceedings of 28th International Conference on Very Large Data Bases, Hong Kong, China*, 2002.

[67] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proceedings of 23rd International Conference on Very Large Data Bases, Athens, Greece*, 1997.

[68] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada*, 1996.

[69] P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA*, 2004.

[70] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of 26th International Conference on Very Large Data Bases, Cairo, Egypt*, 2000.

[71] J. Shanmugasundaram, E. J. Shekita, J. Kiernan, R. Krishnamurthy, S. Viglas, J. F. Naughton, and I. Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. *SIGMOD Record 30(3): 20-26 (2001).*

[72] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, UK*, 1999.

[73] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Database and Expert Systems Applications, 10th International Conference, Florence, Italy*, 1999.

[74] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, 2001.

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

[76] J. S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania, USA*, 1999.

[77] 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, San Diego, California, USA*, 2003.

[78] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Containment Join Size Estimation: Models and Methods. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA*, 2003.

[79] 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, Bangalore, India*, 2003.

[80] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom Histogram: Path Selectivity Estimation for XML Data with Updates. In *Proceedings of the 30th International Conference on Very Large Data Bases, Toronto, Canada*, 2004.

[81] K. L. Wu, S. K. Chen, and P. S. Yu. Interval query indexing for efficient stream processing. In *Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA*, 2004.

[82] X. Wu, M. L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proceedings of the 20th International Conference on Data Engineering, Boston, MA, USA*, 2004.

[83] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *Proceedings of 8th International Conference on Extending Database Technology, Prague, Czech Republic*, 2002.

[84] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural Join Order Selection for XML Query Optimization. In *Proceedings of the 19th International Conference on Data Engineering, Bangalore, India*, 2003.

[85] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: A Path-based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Trans. Internet Techn. 1(1): 110-141 (2001).*

[86] T. Yu, T. W. Ling, and J. Lu. TwigStackList¬: A Holistic Twig Join Algorithm for Twig Query with Not-predicates on XML Data. In *The 11th International Conference on Database Systems for Advanced Applications, DASFAA, Singapore*, 2006.

[87] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Sys-

tems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara, California, USA*, 2001.