Evaluation of Path-oriented Queries in Document Databases

by

Yong Shi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

University of Manitoba
Winnipeg, Manitoba, Canada
September 2006

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES
*****
COPYRIGHT PERMISSION

EVALUATION OF PATH-ORIENTED QUERIES IN DOCUMENT DATABASES

BY

YONG SHI

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

OF

MASTER OF SCIENCE

Abstract

Extensible Markup Language (XML) is emerging as a dominant standard for data representation and data exchange over the Internet. As XML is gaining widespread adoption, it is expected that more and more information will be formatted as XML documents. Managing a large amount of XML documents raises a number of challenges. One of the most important issues is the query evaluations against XML documents by which a database will be retrieved to find all those documents that satisfy a given searching condition. The purpose of this thesis is to develop an efficient method for evaluating path-oriented queries in document databases. Path-oriented queries are the queries submitted for retrieving XML documents from databases. Many traditional query evaluation methods often use some types of index structures, such as path indexes, to evaluate path-oriented queries. These methods fail to recognize that the problem of path-oriented query evaluation is in fact a tree inclusion problem. Therefore, they often perform well on simple queries but fail to do so in case of large and complex queries. In this thesis, I propose a new query evaluation method to eliminate this deficiency. The new approach combines a top-down tree inclusion algorithm with the signature technique to achieve high efficiency.

Acknowledgments

This thesis work was performed under the supervision of my supervisor, Dr. Yangjun Chen. He is a learned professor in his research area, and he honored me with his insights and invaluable suggestions and comments during the research. I am greatly thankful to Professor Chen for his continued guidance, advice, and generous financial support.

I would also like to thank Mr. Donovan Cooke and Mr. Yan Zhuang who spent their time on the performance experiments. I am grateful to their patience and valuable helps.

My deepest gratitude goes to my parents, ZhaoChun Shi and Ning Zhang, my sister Yun Shi for their emotional support and encouragement. I dedicate this thesis to them.

Finally, I want to express all my love to my wife, Jing Zhang. Her support and unconditional love are beyond words. Without Jing this thesis would not have been written. She made me believe I was capable of succeed on this dream. All my love to her.

Contents

Abstract i

Acknowledgments ii

Contents iii

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 XML Document and Document Type Definition (DTD) 2
- 1.2 Problem Statement 4
- 1.3 Preliminaries 6
  - 1.3.1 Tree 6
  - 1.3.2 Tree Inclusion Problem 6
  - 1.3.3 Signatures 8
- 1.4 Objectives 11
- 1.5 Thesis Organization 11

2 Related Work 12
- 2.1 XML Query Evaluation 12
- 2.2 Tree Inclusion Problem 18
  - 2.2.1 Ordered Tree Inclusion 19
  - 2.2.2 Unordered Tree Inclusion 22

3 Path-oriented Query Evaluation 26
- 3.1 Tree Inclusion 27
  - 3.1.1 Top-down Tree Inclusion 27
  - 3.1.2 A Top-down Tree Inclusion Algorithm 32
  - 3.1.3 Correctness of the Algorithm 39

3.2 Signature Techniques 40
3.3 Integration of Signatures and Top-down Tree Inclusion 41
3.4 Signature Tree Technique 43
3.4.1 Signature Tree 44
3.4.2 Constructing a Signature Tree 45
3.4.3 Querying a Signature Tree 50

4 Performance Evaluation **55**
4.1 Implementation 55
4.1.1 Determining the Length of Signatures 56
4.1.2 Mapping XML data into a Relational Database 58
4.2 Experimental Setup 61
4.3 Data Sets 61
4.4 Comparable Methods 64
4.5 Requirements of Data Storage 64
4.6 Experiment on Shakespeare data set 66
4.6.1 Queries 66
4.6.2 Test Method 68
4.6.3 Results 68
4.7 Experiment on DBLP data set 72
4.7.1 Queries 72
4.7.2 Results 77

5 Conclusion and Future Work **79**
5.1 Conclusion 79
5.2 Future Work 80

Bibliography **82**

List of Tables

4.1 A sample of the Element relation 60
4.2 A sample of the Text relation 61
4.3 The characteristics of data sets 62
4.4 Table sizes (MB) of IEW method 65
4.5 Table sizes (MB) of IPW method 65
4.6 Table sizes (MB) of ViST method 65
4.7 Table sizes (MB) of TIS method 65
4.8 Comparison of total table sizes (MB) 65
4.9 Group I. Queries with incremental path lengthes 66
4.10 Group II. Queries with incremental degrees 67
4.11 Group III. Queries matching at higher level of the document 67
4.12 Group IV. Queries matching at middle level of the document 67
4.13 Group V. Queries matching at lower level of the document 67

List of Figures

1.1 An XML document and its tree model 1
1.2 Tree, Subtree, and Forest 7
1.3 An ordered tree inclusion 8
1.4 Signature construction and comparison 10

2.1 Inverted indices on an XML document 13
2.2 Structure encoded sequences 16
2.3 Suffix-tree-like structure for indexing structure encoded sequences . . 17
2.4 False alarm 19
2.5 An algorithm for the approximate tree embedding problem 24

3.1 The case of the observation (3) 28
3.2 A subtree inclusion example 29
3.3 The recursive process of a subtree inclusion 30
3.4 Illustration of the tree inclusion algorithm 37
3.5 Signature a tree 42
3.6 Accelerate evaluation by signatures 44
3.7 A sample of signature tree 45
3.8 Constructing a balanced signature tree 46
3.9 Querying a balanced signature tree 54

4.1 The test of new signature formula 58
4.2 The tree structure of a Shakespeare's play 63
4.3 Execution time of queries in Group One 69
4.4 Execution time of queries in Group Two 69
4.5 Execution time of queries in Group Three 70
4.6 Execution time of queries in Group Four 70
4.7 Execution time of queries in Group Five 71
4.8 Queries of small sizes 73
4.9 Queries of median sizes 74
4.10 Queries of large sizes 75
4.11 Execution time of queries in small sizes 76

4.12 Execution time of queries in median sizes 76
4.13 Execution time of queries in large sizes 77

Chapter 1

Introduction

The eXtensible Markup Language (XML) [3] is a standard format of data representation proposed by W3C (World Wide Web Consortium). Through customized markup tags, XML can describe not only the data itself, but also its semantics, which enables users to organize information with great freedom. Due to its simplicity and flexibility, XML is now a dominant standard for representing and exchanging data over the Internet.

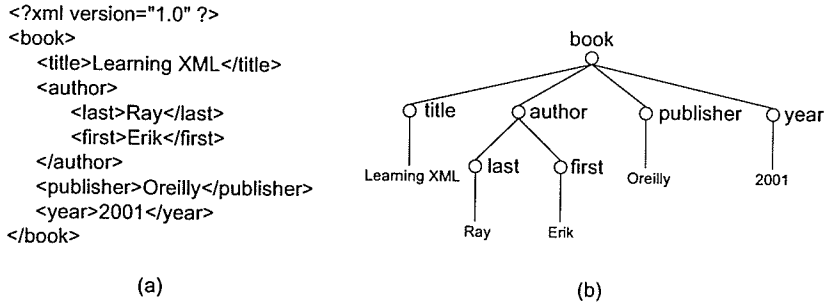


Figure 1.1: An XML document and its tree model

1.1 XML Document and Document Type Definition (DTD)

In Figure 1.1(a), a book record is represented in XML format, which is normally known as an *XML document*, containing three basic components: *Element*, *text* and *attribute*. The element is a piece of text delimited by a pair of tags (called *start tag* and *end tag*) such as <book> and </book> in the above example. The text is the "raw" data that represents the content of a document such as "Learning XML" appearing in Figure 1.1(a). The attribute is a name/value pair that represents the additional properties of an element. In general, an element may contain texts and sub-elements, i.e. multiple elements and texts can be nested in some way. Elements, together with texts, exhibit the hierarchical nature of an XML document. Therefore, any XML document can be modelled as a tree-like structure (we call it a document tree or an XML tree) in which all texts are mapped to the leaf nodes and all elements are mapped to the internal nodes. For example, Figure 1.1(b) shows the tree structure associated with the sample document shown in Figure 1.1(a). Within the XML tree, along a route from the root node to a leaf node, the edges and nodes make up of a *path*.

An XML document must be well-formed in order to be processed correctly. A *well-formed* document is one that has only one root element, has matching start and end tags for every element, has no tags nested out of order, and is syntactically correct in regard to the XML specification. An XML document can be, but not always required to be valid. A *valid* document is one that is well-formed, and also conforms to its Document Type Definition (DTD). A DTD is a context-free grammar that defines

the potential structure of an XML document with a list of legal elements. It specifies what constraints those elements must follow and how those elements are put together. The following is the DTD for our sample XML document shown in Figure 1.1.

```
<?xml version="1.0"?>
<!DOCTYPE book [
    <!ELEMENT book (title, author, publisher, year)>
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (last, first)>
    <!ELEMENT publisher (#PCDATA)>
    <!ELEMENT year (#PCDATA)>
    <!ELEMENT last (#PCDATA)>
    <!ELEMENT first (#PCDATA)>
]>
<book>
    <title>Learning XML</title>
    <author>
        <last>Ray</last>
        <first>Erik</first>
    </author>
    <publisher>Oreilly</publisher>
    <year>2001</year>
</book>
```

In the DTD, the structure of every XML element is declared by which an element with one or more sub-elements are indicated by a list of the names of its sub-elements

(put in a pair of parentheses). The order of these sub-elements appearing in the parentheses must be maintained when they appear in the document. For example, in our sample document, if the sub-elements of "author" element switched their position in the document, which means that "first" element appears before "last" element, the document will be no longer valid according to the DTD.

1.2 Problem Statement

As the XML is gaining a widespread adoption, it is expected that more and more information will be formatted as XML documents. However, managing a large amount of XML documents raises a number of challenges. Among them, the most important issue is the query evaluation against XML documents, by which a database will be retrieved to find all those documents that satisfy a given searching criteria. Retrieving XML documents is not as easy as searching keywords from flat (i.e. non-structured) text files because the hierarchical structure that resides in every XML document should be considered. To this end, several query languages, such as *XML-QL* [11], *XPATH* [9], *XQL* [18], and *XQuery* [1], have been proposed. These languages share a common feature: they allow users to submit XML queries in the form of path expressions to navigate through the tree structure of an XML document. We call this kind of queries *path-oriented queries*. A path expression is used to describe an XML path in a readable format. It is similar to a path in a file management system, but with some extensions (e.g. with some predicate). For example, /book/author/[last = 'Ray'] is a path expression that enquiries one of the paths in the tree shown in Figure 1.1(b) to find any book written by any author whose last name is Ray. Multiple path expressions can form a complex query that contains multiple paths, which is in fact a

tree pattern (we call it a *query tree*). Up to now, most methods proposed to evaluate path-oriented queries make use of various index structures on XML documents such as those discussed in [21, 23]. For instance, in [23], each word (a tag or a text word) in an XML document is numbered according to its position in the document, which is used as an index and stored in database tables. In this way, the evaluation of a path-oriented query can be done by performing a series of join operations. In [21], the indexes are generated for tree paths, which dramatically reduces the number of joins to be performed when evaluating a query.

An alternative method, which has been overlooked in the database research community, is to treat the problem as a tree-inclusion problem although much theoretic research has been done on this issue and several interesting algorithms for checking tree inclusion have been proposed in the literature [5, 13, 14]. However, all of them work in a bottom-up way and assume that the whole document tree and the whole query tree can be accommodated completely in main memory prior to any operation. Obviously, it is not feasible in the case of large volume of data. Therefore, those methods limited their usability for database applications.

In this thesis, I propose a new method for evaluation of path-oriented queries which is based on a new tree inclusion algorithm discussed in [8] and a new index technique called *signature tree* [6]. This new algorithm tackles tree inclusion problems in a top-down fashion and each time manipulates only a small portion of a tree and therefore is well suitable for a database environment. The signature tree technique is a new method of organizing signatures for fast scanning and locating desired signatures. I integrate it into the top-down tree inclusion algorithm and use it as an index structure to immediately find all potential matching documents.

1.3 Preliminaries

In this section, we present some background information related to this thesis, including the concepts of trees, the problem of tree inclusion, and the principle of signature techniques. These knowledge provides a necessary basis for the further discussion.

1.3.1 Tree

Let T be a tree as shown in Figure 1.2(a). We denote its root node by r_T. Let v be any node of T. The children of v are called *sibling* nodes. The number of v's children is called the *degree* of v. For example, node b, c and d in Figure 1.2(a) are siblings. Their degrees are 1, 0 and 2, respectively. A tree, which is a child of v, is called the *subtree* of v. Multiple disjoint subtrees or trees form a *forest*. In Figure 1.2(b), the trees rooted at b, c and d are subtrees of node a, and they are a set of subtrees forming a forest. For any tree, there are three important properties: *size*, *height* and *width*. The tree *size* is the total number of the nodes. The tree *height* is the length of the longest path in the tree from the root to a leaf node. The tree *width* is the number of the leaf nodes. In Figure 1.2(a), the size of T is 7, the height of T is 3, and the width of T is 4.

1.3.2 Tree Inclusion Problem

Let T be a rooted tree. T is a *labelled* tree if each node of T is assigned a character string. Let S and T be rooted, labelled trees. We say that S is *included* in T if S can be obtained by deleting nodes of T. When a node v of T is deleted, the children of v will become the children of the parent of v. Hence, the *tree inclusion problem* is

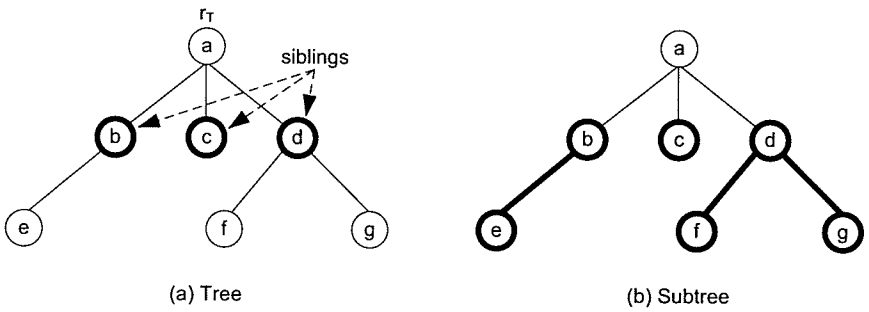


Figure 1.2: Tree, Subtree, and Forest

to determine if S can be embedded in T.

A tree can be ordered or unordered. T is *ordered* if a left-to-right order between two sibling nodes is significant. Let S, T be two rooted, labelled trees. Let $V(S)$ and $V(T)$ be the node sets of S and T respectively. We define an *ordered tree inclusion* (f, S, T) as an injective function $f : V(S) \rightarrow V(T)$ such that for all nodes $v, u \in V(S)$,

- label(v) = label(u); (label preservation condition)
- v is an ancestor of u iff $f(v)$ is an ancestor of $f(u)$; (ancestor condition)
- v is to the left of u iff $f(v)$ is to the left of $f(u)$. (sibling condition)

For example, Figure 1.3 is an example of the ordered tree inclusion. The definition of an *unordered tree inclusion* is the same as above except without the sibling condition. In [13], the unordered tree inclusion problem had been shown to be NP-complete. Therefore, my proposed work only uses ordered tree inclusion method as a basic strategy to evaluate path-oriented queries and adapt it to unordered cases by imposing lexicographic order on both documents and queries.

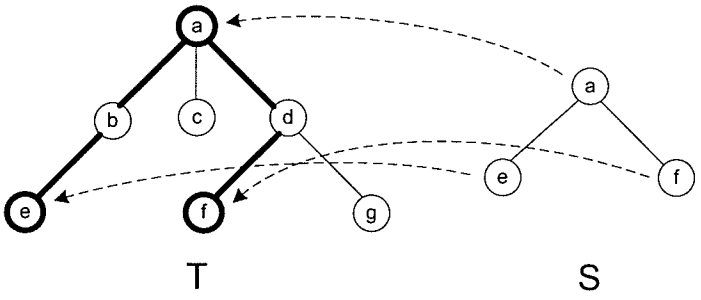


Figure 1.3: An ordered tree inclusion

1.3.3 Signatures

The signature technique [7, 12] was originally introduced as a text indexing methodology. Nowadays, it has been used in a wide range of applications. A *signature* is a binary bit string that represents a word in an abstract format. The main idea of the signature technique is that a document is considered as a set of words. Each word in the document is hashed into a bit string of length F such that exactly $m(\leq F)$ bits are set to 1. The resulting bit string is a *word signature*. The *document signature* s is constructed by superimposing (i.e. bitwise OR, denoted as $\vee$) all the word signatures. Suppose there are N documents. The N document signatures will be stored in a *signature file* sequentially. To search for a word in these documents, the word is hashed by the same function to produce a *query signature* s_q. Then it will be compared (i.e. bitwise AND, denoted as $\wedge$) with each document signature in the signature file to find out where the word is located. Figure 1.4 depicts the signature generation and comparison process. Given a document that contains three words, say "information", "retrieval", and "method". Each word produces a signature of length $F = 12$, in

which $m = 4$ bits are set to 1. They are superimposed together to get a document signature s stored in a signature file. When a query arrives, the document signatures in the corresponding signature file are scanned and many unqualified documents are discarded. The rest are either checked (so that the "false drop" are discarded; see below) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for word signatures. The query signature is then compared to each document signature in the signature file. Three possible outcomes of the comparison are exemplified in Figure 1.4: (1) the document matches the query; that is, for every bit set in s_q, the corresponding bit in the document signature s is also set (i.e. $s \wedge s_q = s_q$) and the document really contains the query word; (2) the document does not match the query ($s \wedge s_q \neq s_q$); and (3) the signature comparison indicates a match but the document in fact does not match the search criteria (i.e. *false drop*). In order to eliminate false drops, the document must be examined after the document signature signifies a successful match.

After superimposing multiple word signatures into a document signature, the density of '1' bits in the document signature may increase, leading to a worse selectivity, which means more false drops would happen. As shown in Figure 1.4, each word signature has only 4 bits set to 1. After superimposing three word signatures into a document signature, the document signature has 9 bits set to 1, which is more than doubled. To retain the selectivity of a document signature (i.e. the ability of filtering unqualified words), a longer signature should be used. However, a signature length is not unlimited. Therefore, an effective method [12] is to divide a document into a number of blocks where each block contains roughly the same number of words.

Word:		Signature:
information		010000100110
retrieval		100010010100
method	v	010100011000
Document signature :		110110111110

Query:	Query signature:		Document signature:			Result:
information	010000100110	∧	110110111110	=	010000100110	Match
XML	011000100100	∧	110110111110	=	010000100100	No match
database	110100100000	∧	110110111110	=	110100100000	False drop

Figure 1.4: Signature construction and comparison

We can construct a block signature for each block by superimposing all of its word signatures. In this way, a single document signature is replaced by multiple block signatures in a signature file. Since the number of words in a block is much less than the number of words in a document, the density of '1' bits in the block signature is lower, thus it achieves a good selectivity. In this method, we can use the following formula [12] to determine the length of signatures properly.

$$F \times \ln 2 = m \times D \tag{1.1}$$

where F is the signature length to be determined, m is the number of bits set to 1 in a word signature, and D is the number of words in a block.

1.4 Objectives

The main goal of this thesis is to create a new query evaluation method for processing path-oriented queries over relational databases, including:

- Implementing a new top-down tree inclusion algorithm which can be applied to efficiently determine whether one tree can be embedded in another.
- Investigating the effectiveness of the signature technique in this problem.

1.5 Thesis Organization

The remain of the thesis is organized as follows. We proceed by first discussing related work in the next chapter. Then, we discuss typical XML query evaluation methods, and techniques solving tree inclusion problem. In Chapter 3, we describe the details of our path-oriented query evaluation method, including the tree inclusion algorithm and the signature technique. Implementation details and the analysis of the experiments are presented in Chapter 4. Chapter 5 concludes the thesis and outlines future work.

Chapter 2

Related Work

This thesis crosses over two related fields: XML query evaluation and tree inclusion problem. In this chapter, we present some related work of both areas.

2.1 XML Query Evaluation

XML query evaluation has attracted a lot of attentions since XML became a universal format for data representation and data exchange. So far most work done on this subject views XML data as a collection of text documents with additional tags. The common idea of those approaches is to index XML documents, and retrieve documents based on the established indices.

In [23], Zhang et al. studied the problem of how to evaluate the containment queries, which is a class of XML queries that evaluate containment and proximity relationships among elements, attributes and texts. For example, the path expression /book/author//'Ray' has two containment relationships. One is book/author; and the other is author//'Ray'. In this path expression, '/' represents a parent-child

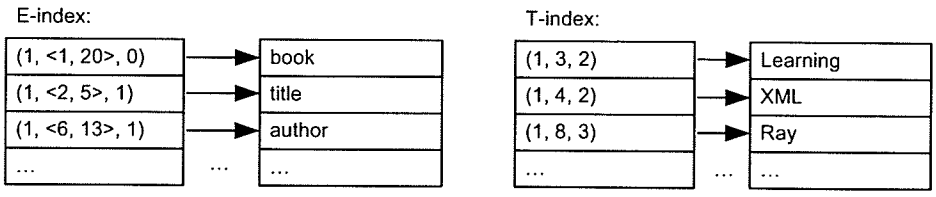


Figure 2.1: Inverted indices on an XML document

relationship and '//' represents an ancestor-descendant relationship. To evaluate such a query, a method based on *inverted lists* [19] is developed in [23], by means of which two kinds of inverted lists are constructed to index elements and texts within a set of documents, including the document number in which they appear, as well as their positions in the corresponding documents. Concretely, two tables of the following form will be defined.

- (*docID*, *wPosition*, *level*) for a text word
- (*docID*, *ePosition*, *level*) for an element

where *docID* is the document number, *wPosition* is the position of a text within a document, *ePosition* is a pair of values: $< s, e >$, representing the start position and the end position of an element, and *level* is the depth of a node position with respect to the root. For instance, Figure 2.1 shows two inverted indexes established for the sample document shown in Figure 1.1. The index for texts is called *T-index*, and the index for elements is called *E-index*.

Let (d, x, l) be an index entry for an element a, and let (d', x', l') be an index entry for a word b. Then, a contains b iff $d = d'$ and $x.s < x' < x.e$. Let (d'', x'', l'') be an index entry for another element c. Then, a contains c iff $x.s < x''.s$ and $x.e > x''.e$.

Using these properties, some simple path-oriented queries can be evaluated. For instance, to process the query: /book/author[last = 'Ray'], the inverted lists of 'book', 'author', 'last', 'Ray' will be retrieved and then their containment relationships will be checked according to the above properties. In a relational database, *E-index* and *T-index* are mapped into the following two relations (the primary keys are underlined):

- **E-index** (element, docno, begin, end, level)
- **T-index** (word, docno, wordPosition, level)

The above index structures are efficient for simple cases, such as whether a word is contained in an element. However, in the case that a query is a non-trivial tree, the evaluation based on these index structures is an exponential time process. To see this, consider the query: /book/author[last = 'Ray'] once again. To evaluate this query, three joins have to be performed. They are the self-joins on *E-index* relation to connect 'book' and 'author', 'author' and 'last', as well as the join between *E-index* and *T-index* relations to connect 'last' and 'Ray'. In general, for a document tree with n nodes and a query tree with m nodes, the checking of containment needs $O(n^m)$ time using this method.

The above method is improved by Seo et al. [21] by introducing indexes on paths to reduce the number of joins as well as the sizes of relations involved in a join operation. This is achieved by establishing four relations to accommodate the inverted lists (the primary keys are underlined):

- **Path** (path, pathID)
- **PathIndex** (pathID, docno, begin, end)

- **Word** (word, wordID)
- **WordIndex** (wordID, docno, pathID, position)

In this way, the number of joins is dramatically decreased. For example, to process the previous query, only two joins are needed. The first join is between the *Path* and *WordIndex* relations with the following join condition:

- *Path*.path = 'book/author/last'
- *Path*.pathID = *WordIndex*.pathID.

The second join is between the result R of the first join and the *Word* relation with the following join condition:

- *R*.wordID = *Word*.wordID
- *Word*.word = 'Ray'

In general, the query evaluation based on such an index structure needs k joins, where k is the number of the words appearing in a query. However, such a time improvement is at cost of memory space since in *Path* relation the element names are repeatedly stored. Concretely, for a document with n nodes, the size of *Path* relation is on the order of $O(n^2)$. Therefore, the time complexity of this method is $O(k{\cdot}l{\cdot}n^2)$, where l stands for the average length of the paths.

To avoid expensive join operations, Wang et al. devised a new index method named ViST [22] for searching XML documents, which is based on a new type of index structures called structure-encoded sequences. Each XML document is transformed into a sequence; so is each query. For example, Figure 2.2 shows two sequences

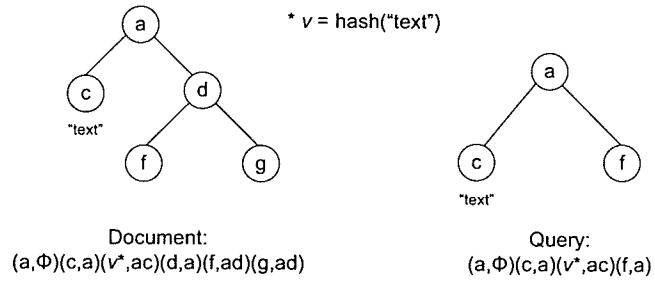


Figure 2.2: Structure encoded sequences

representing a document and a query, respectively. The structure-encoded sequences encode both structure and content of the XML data. Hence, evaluating an XML query is in a way similar to a subsequence (non-contiguous) matching. Since an XML query can be answered as a tree structure without being disassembled into multiple subqueries (i.e. paths), join operations are avoided.

They used a suffix-tree-like structure to index document sequences like the one shown in Figure 2.3, which includes three documents. Each tree node in such structure holds two pairs of index values: a $< symbol, prefix >$ pair and a $< preorder, size >$ pair. The $< symbol, prefix >$ pair represents the ancestor-descendant relationships of the nodes in the original document tree called D-Ancestorship, where the symbol is the label of a document tree node and the prefix is the path from the root down to the node. The $< preorder, size >$ pair represents the ancestor-descendant relationships of the nodes in the suffix-tree-like structure called S-Ancestorship, where the preorder is the prefix traversal order of a node in the suffix-tree-like structure and the size is the total number of descendants of this node in this structure. Suppose x and y are

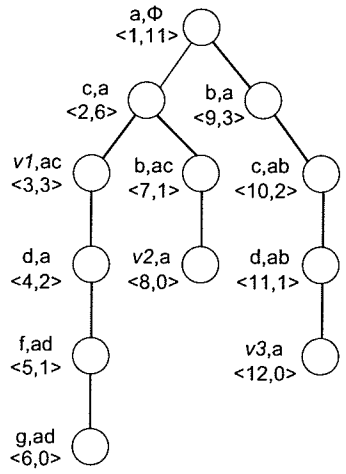


Figure 2.3: Suffix-tree-like structure for indexing structure encoded sequences

labelled $< n_x, size_x >$ and $< n_y, size_y >$ respectively, node x is an S-Ancestor of node y iff $n_y \in \{n_x + 1, n_x + 2, \ldots, n_x + size_x\}$.

Given a query sequence $q_1, \ldots, q_n$, their objective is to sequentially match each query node to a document node not only satisfying the D-Ancestorship but also satisfying the S-Ancestorship. Due to the nature of non-contiguous sequential searching, this matching process is extremely costly. Therefore, they introduced two kinds of B^+trees to speed up the process. A D-Ancestorship B^+tree is constructed using nodes' $< symbol, prefix >$ pairs as keys. Since multiple documents may have nodes with the same pair of $< symbol, prefix >$, each $< symbol, prefix >$ pair is associated with a S-Ancestorship B^+tree which is constructed using those nodes' preorder numbers as keys. Based on these two B^+trees, they can perform a subsequence matching as follows. Suppose x, labelled with $< n_x, size_x >$, is the node matching of a query

node q_{i-1} in the query sequence $q_1, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n$. To match the next node q_i, they first use q_i's $< symbol, prefix >$ pair to query the D-Ancestorship B^+tree to find a corresponding S-Ancestorship B^+tree. Within this tree, they then issue a range query $n_x < n \leq n_x + size_x$ to find all the descendants of x. For each descendant, they use the same process to match the next query node q_{i+1} until they reach the last node in the query sequence. Finally, all the documents which are associated with the last matching document node are the answer to the query.

With the help of B^+trees, the time complexity of this sequence-based method is $O(\sum_{i=1}^{m} K_i \log(n))$, where K_i is a recursive factor, m and n are the number of nodes in a query tree and a document tree. Thus, it gains some performances for certain queries. However, it still has several disadvantages. For querying a document tree whose elements have many identical labels and paths, the search would become very inefficient because of many recursions. Also, in the case of a large data set, the whole construction process of structure encoded sequences is much expensive. Moreover, due to the problem of query equivalence (i.e. a subsequence matching does not imply a successful retrieval), the method could bring some false alarms into final results. For example, in Figure 2.4, there is a subsequence matching between the document and the query. However, the document does not in fact match the query pattern.

2.2 Tree Inclusion Problem

The tree inclusion problem was originally introduced by Knuth in [15] as an exercise. Motivated by solving the problem of querying structured text databases, Kilpeläinen first began to study the problem. In his Ph.D thesis [13], he presented a detailed taxonomy, which classified the problem into two broad categories: ordered and unordered

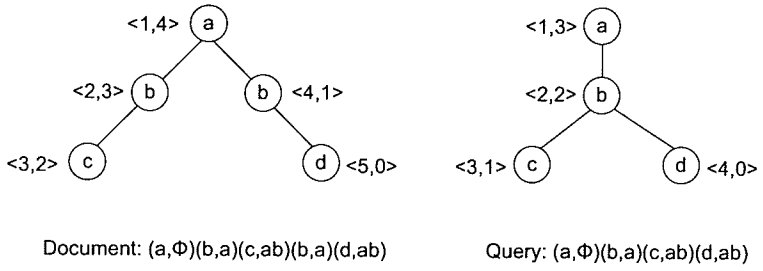


Figure 2.4: False alarm

with each containing five problems: tree inclusion, path inclusion, region inclusion, child inclusion and subtree problem. He suggested that each problem in unordered category is a special case of its previous one and each problem in ordered category is a special case of its corresponding unordered version. He proved that unordered tree inclusion is an NP-complete problem.

2.2.1 Ordered Tree Inclusion

Kilpeläinen and Mannila [13, 14] have presented the first polynomial time algorithm for solving ordered tree inclusion problem. It uses $O(|Q|\cdot|D|)$ time and space, where $|Q|$ and $|D|$ are the numbers of nodes in a query tree and a document tree, respectively.

The main idea of their algorithm can be summarized as follows. Let $T(r)$ be an ordered, rooted tree with root node r and subtrees $r_1, \cdots, r_i$. $V(T)$ be the node set of $T(r)$. The *left relatives* $lr(v)$ and the *right relative* $rr(v)$ of a node $v \in V(T)$ are

defined as follows

$$lr(v) = \{x \in V(T) \mid pre(v) > pre(x) \wedge post(v) > post(x)\}$$

$$rr(v) = \{x \in V(T) \mid pre(v) < pre(x) \wedge post(v) < post(x)\}$$

, where $pre(v)$, $post(v)$ stand for the preorder number and the postorder number of node v, respectively. Let Q and D be two ordered labelled trees. A *root-preserving embedding* $R(Q, D)$ is an embedding (i.e. inclusion) (f, Q, D), where $f(r_Q) = r_D$. To determine if D can include Q, the algorithm constantly searches for root-preserving embedding by processing the subtrees of Q from left to right, trying to embed them as deep and as left in D as possible. However, the algorithm may need exponential time to check such embedding. To this end, they introduce a concept called *left embedding.* Let $F_Q = (Q_1, \cdots, Q_k)$ be a forest, and let ε be a collection of embedding of F_Q in a forest F_D. An embedding $f \in \varepsilon$ is a left embedding of ε if $post(f(r_{Q_k})) \leq post(g(r_{Q_k}))$ for every $g \in \varepsilon$. Their objective is to look for left embedding between the subtrees of Q and D. Hence, they construct a $|Q| \times |D|$ table e with each entry $e(u, v)$ in the table, (where $u \in V(Q)$ and $v \in V(D)$) defined as follows

$$e(u, v) = \min(\{x \in rr(v) \mid \exists f \in R(Q(u), D(v))\} \cup \{d + 1\}),$$

where $d = |D|$. That is, $e(u, v)$ is the closest right relative of v which has a root-preserving embedding of $Q(v)$, i.e. the node with next left embedding of $Q(u)$ in $D(v)$. The algorithm uses dynamic programming to compute every table entry $e(u, v)$. For each u, the algorithm traverses the nodes of D in postorder using a pointer p. At each node $v \in V(D)$, it checks if there is a root-preserving embedding of $Q(u)$ in $D(v)$. If not, the entry $e(u, p)$ is set to $d + 1$, otherwise $e(u, p)$ is set to v for all $p \in lr(v)$. D includes Q if and only if $e(q, 0) \leq d$ where $q = |Q|$.

In the case of large trees, the $O(|Q| \cdot |D|)$ time and space complexities may be unacceptable. Hence, Kilpeläinen [13] also presents a more space-efficient algorithm to improve the space usage, which requires $O(|Q| \cdot depth(D))$ space, where $depth(D)$ is the tree depth. The idea of the algorithm is to use a set of *sibling intervals* that is marked by a start number and an end number to represent a sequence of sibling nodes $< v_1, v_2, \cdots, v_k >$ in Q, which can be embedded in D if there is an ordered inclusion of $< Q(v_1), Q(v_2), \cdots, Q(v_k) >$ in D, where $Q(v_i)$ is the subtree rooted at v_i, $(1 \leq i \leq k)$. For each node w of D, the algorithm uses a bottom-up approach to compute a match set $M(w)$ consisting of sibling intervals that can be embedded in $D(w)$ (i.e. the subtree rooted at w). If the sibling interval of r_Q appears in the last result, Q is embedded in tree D. The space is saved by merging two sets of sibling intervals into one set of sibling intervals to make it *simple*, which means that it does not contain two distinct members one of which contains the other. A merge-like procedure is used to merge two lists of sibling intervals in which members are sorted by their start numbers.

Chen [5] improves the above algorithms by introducing a concept called *shell* and a *ϱ-lists* data structure to further reduce the number of embedding checks. The idea is to apply a more compact format to represent each set of sibling intervals in Q in which the subtrees of the sibling intervals form a maximal forest. To this end, he uses the following set of triples to describe the forests of a tree Q, where d_v is the degree of the node v

$$\tilde{Q} \equiv \{(v, a, b) \mid v \in V(Q), 1 \leq a \leq b \leq d_v\}.$$

For $q = (v, a, b) \in \tilde{Q}$, $forest(q)$ denotes the subforest of Q that consists of subtrees $tree(v[a]), tree(n[a+1]), \cdots, tree(n[b])$, where $n[i]$ is the ith child of node v. Let s,

$t \in \tilde{Q}$. s is said to be less than or equal to t (denoted $s \preceq t$) if $forest(s)$ is fully covered by $forest(t)$. Given $S \subseteq \tilde{Q}$, the *shell* of S is the set $S^o = \{s \in S \mid \exists s' \in S$ such that $s' \preceq s\}$. S is *reduced* if $S = S^o$. To speed up the generation of a reduced set S^o, the set $S = \{s_1, \cdots, s_k\}$ is organized as a ϱ-list $< s_1, \cdots, s_k >$ in which $\varrho(s_1) \leq \varrho(s_2) \leq \cdots \varrho(s_k)$, where $\varrho(s_i) \equiv pre(v[a]), 1 \leq i \leq k$, for $s = (v, a, b) \in \tilde{Q}$. The algorithm runs $O(|l(Q)| \cdot |D|)$ time and $O(|l(Q)| \cdot min(depth(D), |l(D)|))$ space, where $|l(Q)|$ and $|l(D)|$ are the number of leaves in Q and D, respectively.

From the above discussion, we see that a bottom-up approach is mainly used in those algorithms. With this approach, the algorithms can recursively build up match sets for parents by scanning their children. However, the algorithms have to have the complete knowledge of tree structures. Thus, the entire trees have to be loaded in main memory for processing. In addition to the drawback, all the bottom-up algorithms are not able to integrate with signatures for optimization due to their bottom-up computation property. Therefore, compared with our approach, those algorithms are not quite suitable for XML query evaluations.

2.2.2 Unordered Tree Inclusion

So far, no algorithm can solve the unordered tree inclusion problem in polynomial time. Unexceptionally, the method proposed by Schlieder and Naumann [20] still requires exponential time, by means of which the problem of evaluating an XML query against a collection of XML documents is reduced to an unordered tree inclusion problem. This approximately embeds an query tree into a document tree such that only the labels and ancestor-descendant relationships of the nodes are preserved. Since the order of siblings is not considered, multiple data nodes (nodes in a document

tree) may share the same label. So the embedding between the query tree and the document tree is not unique, that is, multiple subtrees of the document tree can be mapped to the same query tree. These subtrees is a set of embeddings approximately matching the query. To find a closest one among all such embeddings, Schlieder et al. extended the above problem to an optimization problem by introducing a cost model to the embeddings. Under this model, each data node d is assigned a *deletion cost* denoted as $cost(d)$. The *embedding cost* C is the sum of the deletion costs of all the data nodes that must be skipped to embed the query. The embedding root d along with its cost C are defined as a *match* $m = (d, C)$.

The algorithm is based on dynamic programming. It processes the query tree bottom-up and incrementally constructs embeddings for a query node q (a node in a query tree) and a data node d until reaching the root of the query tree. For each query node, only the data nodes that have the same label as the query node are processed. For each matching data node d, the algorithm tries to embed the query subtree rooted at q in the document subtree rooted at d, that is, constructs all the possible embeddings for the query subtree rooted at q. Each of those embeddings is represented by a match. To construct the match set of the current query node q, the algorithm builds all the combinations of the match sets belonging to the children of q, which are the Cartesian products of those match sets, leading to an exponential time complexity. Since not all the combinations represent potential embeddings, only *proper combinations* are chosen. This can be done as follows, for each combination S, the algorithm checks whether all the data nodes of the matches in S are descendants of the data node d and whether the matches in S are not *blocking*, i.e. any two matches do not share common nodes. In addition, the *preorder* numbers of the data

nodes are used to verify ancestor-descendant relationships and blocking matches in the construction of a proper combination. During the above process, the algorithm builds a match set for each query node by combining the matching data node with the proper combinations belonging to its children. Within each match set, only the match with the minimal cost (called a *minimal match*) is selected. Finally, when the algorithm reaches the query root, the set of all minimal matches selected so far is the result of the algorithm.

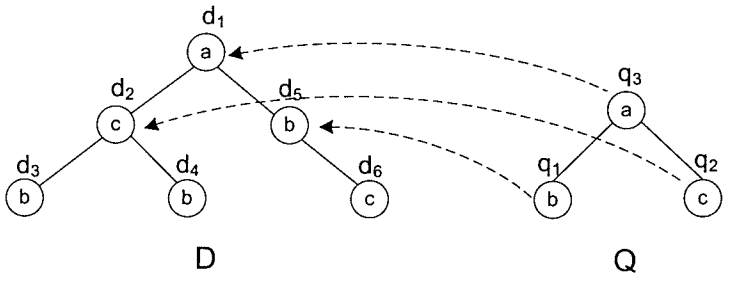


Figure 2.5: An algorithm for the approximate tree embedding problem

For example, consider the document tree D and the query tree Q in Figure 2.5. The algorithm processes the query nodes $Q = \{q_1, q_2, q_3\}$ in postorder. Since both q_1 and q_2 are leaf nodes, their match sets are just the sets of matching data nodes. For q_1, the match set is $M_{q_1} = \{d_3, d_4, d_5\}$. For q_2, the match set is $M_{q_2} = \{d_2, d_6\}$. Now, the algorithm is processing the query root q_3. It first builds all the combinations of the match sets belonging to the children of q_3, that is, the Cartesian product $M_{q_1} \times M_{q_2} = \{\{d_3, d_2\}, \{d_3, d_6\}, \{d_4, d_2\}, \{d_4, d_6\}, \{d_5, d_2\}, \{d_5, d_6\}\}$. To choose the proper combinations from all the combinations, the algorithm then check for each combination whether it qualifies. Since the matches $\{d_3, d_2\}$, $\{d_4, d_2\}$ and $\{d_5, d_6\}$ are blocking, the proper combinations are $\{d_3, d_6\}$,$\{d_4, d_6\}$ and $\{d_5, d_2\}$. Lastly, the

algorithm selects the match with the minimal cost, in this case, $\{d_5, d_2\}$ is the minimal match. Combining it with the matching data node d_1, the final result of this process is a set of minimal matches $M_{q_3} = \{d_1, d_2, d_5\}$.

The runtime complexity of this algorithm is $O(|Q| \cdot I \cdot s^{k+1} \cdot k \cdot (k+h))$, where $|Q|$ is the number of query nodes, I is iteration cycles that need to fetch matching data nodes, s is the number of match data nodes for each query node, k is the number of children of each query node and h is the height of the document tree. Due to the factor s^{k+1}, the complexity of the algorithm remains exponential. From the above description, we can see that the algorithm works in a bottom-up way, and is not able to work with the signature technique in a top-down fashion to speed up evaluation process. This is a major shortcoming when it is compared with our evaluation method.

Chapter 3

Path-oriented Query Evaluation

This chapter describes the method that I implemented to evaluate path-oriented queries in a document database. The main purposes of this method are: 1) to efficiently retrieve all matching documents from the database for a given query; 2) to avoid expensive join operations which many index-based methods have suffered. To achieve these purposes, we devise a method based on two basic techniques: the tree inclusion algorithm and the signature technique. The tree inclusion algorithm treats every document and query as trees, and checks the query tree against each document tree to find out whether the query tree can be successfully embedded in it. The signature technique helps to speed up the process of tree inclusion by: 1) dramatically reducing the number of documents that the tree inclusion algorithm needs to check; 2) eliminating all unnecessary subtree inclusion calls. By combining these two powerful techniques, we are able to efficiently find all the documents matching a given query without involving any join operations.

3.1 Tree Inclusion

As pointed out by Mannila and Räihä [17], the evaluation of path-oriented queries is in essence a tree inclusion problem. Therefore, the tree inclusion algorithm is the core of our query evaluation method. Since there is no polynomial time solution that exists for unordered tree inclusion problem, we implemented only a tree inclusion algorithm that targeted for ordered tree inclusion problem. For unordered query evaluation, we can simply enforce an ordering among sibling nodes. The DTD (Document Type Definitions) or XML Schema for an XML document contains information about a linear order of all elements defined in the document. Hence, we can use it as a pre-defined order that both document and query have to follow. If the DTD or the XML Schema is not available, we simply use the lexicographical order of the element labels. In addition, we can determine the ordering by checking the signatures of tree nodes.

3.1.1 Top-down Tree Inclusion

One of the main problems that any bottom-up tree inclusion algorithm suffers is lack of supporting optimization techniques such as signatures, inverted files, or any kind of indexing techniques. We avoid the problem by designing the tree inclusion algorithm working in a top-down fashion. Suppose that D and Q are trees representing a document and a query respectively, and D includes Q. Then, we have the following three observations.

1. Let r_D and r_Q be the roots of tree D and Q respectively. If $label(r_D) = label(r_Q)$, i.e. there is a root-preserving embedding, then we must check if the subtrees of r_D can include all the subtrees of r_Q.

2. Let $D_1, ..., D_k$ be the subtrees of r_D. Let $Q_1, ..., Q_l$ be the subtrees of r_Q. If $label(r_D) = label(r_Q)$, there must exist two sequences of integers: $k_1, \cdots, k_j$ and $l_1, \cdots, l_j$ and $(l_j \leq l)$ such that D_{k_i} includes $< Q_{l_{i-1}+1}, \cdots, Q_{l_i} >$ $(i = 1, \ldots, j, l_0 = 0, l_j = l)$, where $< Q_{l_{i-1}+1}, \cdots, Q_{l_i} >$ represents a forest that contains subtrees $Q_{l_{i-1}+1}, \cdots,$ and Q_{l_i}.

3. If $label(r_D) \neq label(r_Q)$, we need to check if there exists a D_i $(1 \leq i \leq k)$ that includes the whole Q (i.e. one of subtrees of r_D).

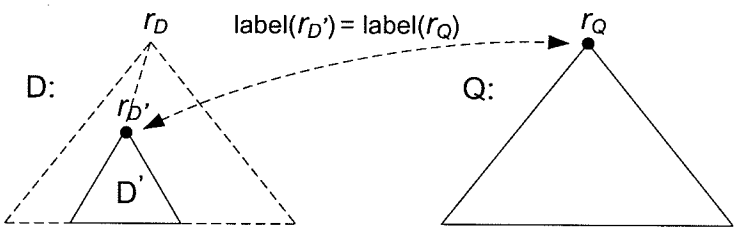


Figure 3.1: The case of the observation (3)

The above observations indicate a top-down approach to find a tree inclusion. The approach consists of two steps. The first step is to anchor a matching position of r_Q in D, which is to find a subtree D' whose root has the same label as r_Q (i.e. $label(r_Q) = label(r_{D'})$). The possible matching location could be r_D (this is the case of our observation (1) in which the tree D' is just the tree D) or $r_{D'}$ (this is the case of our observation (3) in which the tree D' has to be one of the subtrees of D, as shown in Figure 3.1). Once the root node r_Q settles down, the next step is to find the tree inclusions of all its subtrees. In the following discussion, we assume that the matching position of r_Q in D is r_D. According to the observation (2), we first need to check D_1 against $< Q_1, \cdots, Q_l >$ to find whether there exists an i such

that D_1 includes $< Q_1, \cdots, Q_i >$. If such i does not exist (i.e. $i = 0$), it means that D_1 does not include all subtree in $< Q_1, \cdots, Q_l >$. Then, we have to check D_2 against $< Q_{i+1}, \cdots, Q_l >$, and so on. The result of this process is that we found two sequences of integers k_i $(1 \leq i \leq j)$ and l_i $(1 \leq i \leq j)$, which make D_{k_1} include a forest $< Q_1, Q_2, \cdots, Q_{l_1} >$, D_{k_2} include a forest $< Q_{l_1+1}, Q_{l_1+2}, \cdots, Q_{l_2} >$, $\cdots$, D_{k_j} include a forest $< Q_{l_{j-1}+1}, Q_{l_{j-1}+2}, \cdots, Q_{l_j} >$. If the last integer of the sequence $l_1, l_2, \ldots, l_j$ is smaller than l (i.e. $l_j < l$), it means that D can not include Q. Otherwise, if $l_j = l$, it means that D includes Q. For example, in Figure 3.2, r_D has four subtrees

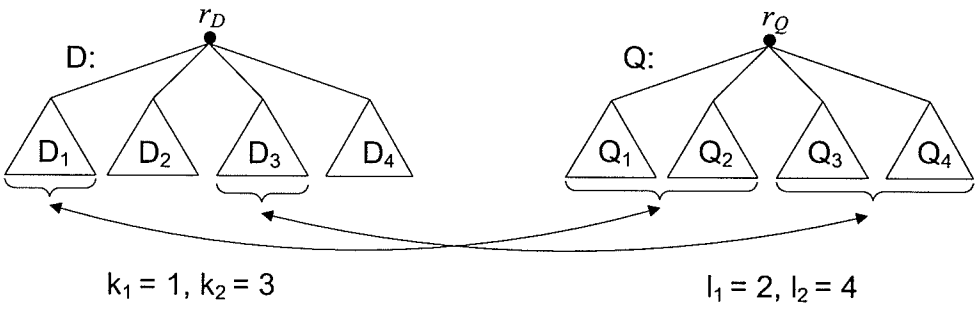


Figure 3.2: A subtree inclusion example

$\{D_1, D_2, D_3, D_4\}$, and r_Q also has four subtrees $\{Q_1, Q_2, Q_3, Q_4\}$. Suppose that D_1 includes $< Q_1, Q_2 >$, and D_3 includes $< Q_3, Q_4 >$. Then, based on the observation (2), we will find two sequences of integers k_i and l_i, which $k_1 = 1, k_2 = 3$ and $l_1 = 2, l_2 = 4$. Since $l_2 = l = 4$, D includes Q in this example. Notice that the sequence of integer k may not be continuous since not all of subtrees in D participate in the inclusions. The above discussion also applies to other situations in which the matching position of r_Q in D is not r_D.

When we check a subtree inclusion such as checking D_1 against $< Q_1, \cdots, Q_l >$, we apply a recursive process to the checking process (see Figure 3.3). Let $D_1, D_2, \cdots, D_k$

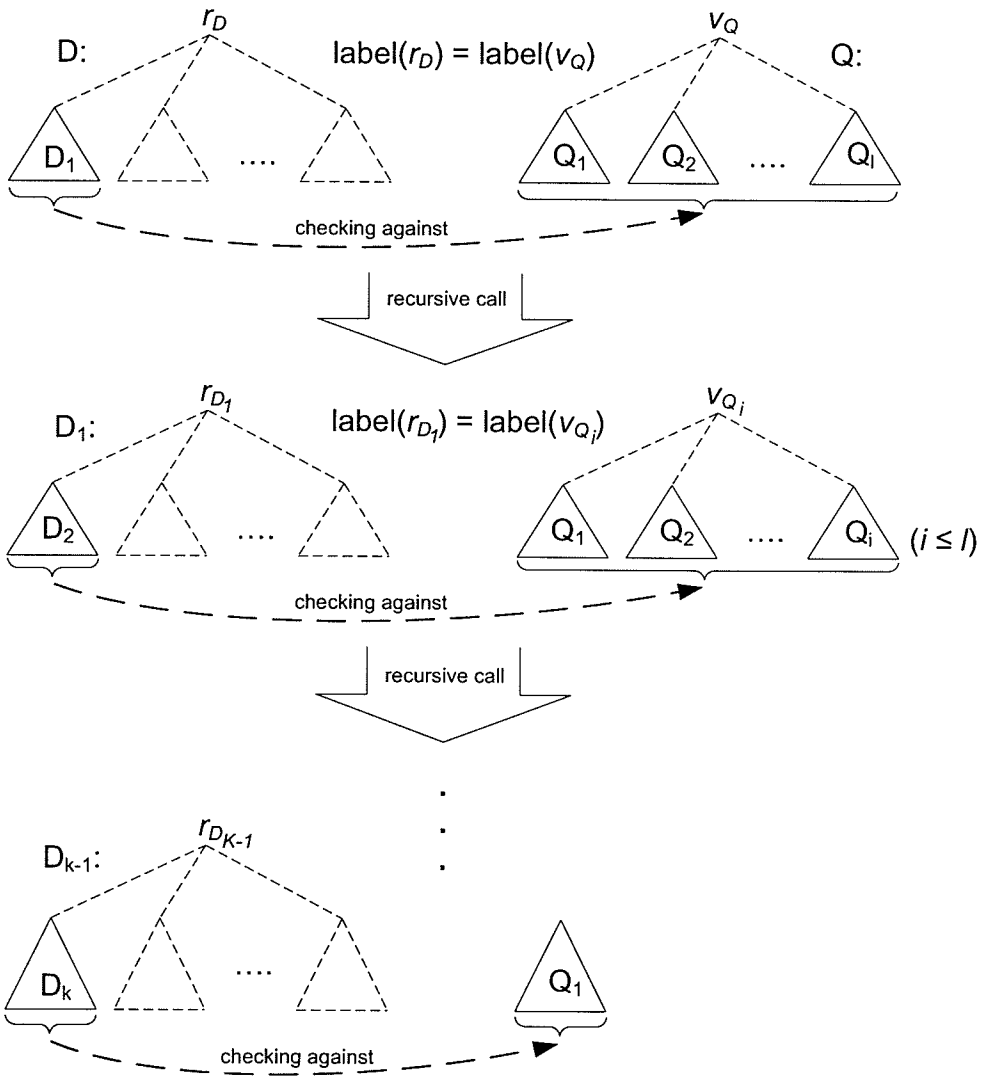


Figure 3.3: The recursive process of a subtree inclusion

be a series of subtrees that are the leftmost subtrees of D, and D_2 is the subtree of r_{D_1}, D_3 is the subtree of r_{D_2}, $\cdots$, D_k is the subtree of $r_{D_{k-1}}$. Since this is an ordered tree inclusion, the "leftmost" is to ensure the ordering. For any forest, we use a virtual node to serve as a temporary root such that a forest can be transformed to a tree with a virtual root. The virtual root can match any node label. For example, v_Q is a virtual root of the forest $< Q_1, \cdots, Q_l >$, which matches the label of r_{D_1}. Then, a new subtree inclusion between D_2 and $< Q_1, \cdots, Q_i >$ is invoked recursively, where the i is an integer such that $|D_2| \geq | < Q_1, \cdots, Q_i > |$ but $|D_2| < | < Q_1, \cdots, Q_{i+1} > |$. Again, we construct a virtual root v_{Q_i} for $< Q_1, \cdots, Q_i >$, which matches the label of r_{D_2}. Then, another new subtree inclusion process is invoked. This type of recursion continues until we check D_k against Q_1 (both D_k and Q_1 could be leaf nodes). If the label of r_{D_k} matches the label of r_{Q_1}, the process returns 1 indicating a successful inclusion, otherwise, it returns 0 indicating a failure. Either of these two results will cause the recursive process to return. During the return, a series of new subtree inclusions may begin, hence, the whole computation process is a top-down process with a bottom-up checking interleaved.

Based on the previous observations, we can devise a computation process as below. Firstly, we compare the root labels of query tree Q and document tree D. If label(r_Q) = label(r_D), we then recursively call the same function to check whether D_1 includes $< Q_1, \cdots, Q_l >$. The process returns an integer i indicating that D_1 includes $< Q_1, \cdots, Q_i >$. If $i > 0$, then we can continue to check if D_2 includes $< Q_{i+1}, \cdots, Q_l >$. Otherwise, if $i = 0$, it indicates that no subtrees of D_1's root includes any of subtrees in $< Q_1, \cdots, Q_l >$. In this case, we need to check whether D_1 includes Q_1. The reason is that although no subtrees of D_1's root includes any subtree in $< Q_1, \cdots, Q_l >$,

D_1 may include Q_1. If D_1 includes Q_1, i will be changed to 1; otherwise, it remains 0. However, if label(r_D) $\neq$ label(r_Q) at first, we have to search for any subtree in $< D_1, \cdots, D_k >$ to include the whole Q. The algorithm repeats the above process until it find an integer i $(= k)$ such that D_i contains all the remaining subtrees of r_Q, or find that no such i exists.

3.1.2 A Top-down Tree Inclusion Algorithm

Algorithm 1 shows the details of how to perform a tree inclusion. The algorithm is a recursive function which initially accepts two parameters: a document tree D, a query tree Q (strictly speaking, they are the root of a document tree r_D and the root of a query tree r_Q), and it returns an integer to indicate the result of a tree inclusion (e.g. the 0 indicates that the document tree can not include the query tree; the 1 indicates a successful tree embedding).

Algorithm 1 TreeInclusion(D, Q)

1. **if** $|D| < |Q|$ **then**
2. **if** Q is a forest $(Q_1, \ldots, Q_q)$ **then**
3. $Q \leftarrow (Q_1, \ldots, Q_i)$ //Looking for i so $|(Q_1, \ldots, Q_i)| \leq |D| < |(Q_1, \ldots, Q_{i+1})|$
4. **else**
5. **return** 0
6. **end if**
7. **end if**
8. $r_D \leftarrow$ root of D

9. $r_Q \leftarrow$ root of Q //If Q is a forest, r_Q will be a virtual root, which matches any label
10. Let $(D_1, \ldots, D_d)$ be the subtrees of D
11. Let $(Q_1, \ldots, Q_q)$ be the subtrees of Q
12. **if** label(r_D) = label(r_Q) **then**
13. **if** r_D is a leaf and r_Q is not a virtual root **then**
14. **return** 1
15. **else if** r_D is a leaf and r_Q is a virtual root **then**
16. **return** 0
17. **end if**
18. $temp \leftarrow (Q_1, \ldots, Q_q)$
19. $i \leftarrow 1$ //i indicates which subtree of D will be used.
20. $j \leftarrow 0$ //j records how many subtrees of Q are included.
21. $x \leftarrow 0$ //x is a temporary variable.
22. **while** $i \leq d$ and $temp \neq \phi$ **do**
23. $x \leftarrow$ TreeInclusion$(D_i, temp)$
24. **if** $x > 0$ **then**
25. $temp \leftarrow temp/(Q_{j+1}, \ldots, Q_{j+x})$ //Matched subtrees are excluded.
26. **else**
27. **if** label(r_{D_i}) = label$(r_{Q_{j+1}})$ **then**
28. $x \leftarrow$ TreeInclusion(D_i, Q_{j+1})
29. $temp \leftarrow temp/(Q_{j+x})$
30. **end if**
31. **end if**

```
32.      i ← i + 1
33.      j ← j + x
34.    end while
35.    if temp ≠ 0 then
36.      if r_Q is a virtual root then
37.        return j
38.      else
39.        return 0
40.      end if
41.    else
42.      if r_Q is a virtual root then
43.        return q
44.      else
45.        return 1
46.      end if
47.    end if
48. else //label(r_D) ≠ label(r_Q)
49.    for i = 1 to d do
50.      x ← TreeInclusion(D_i, Q)
51.      if x = q then
52.        return x
53.      end if
54.    end for
55.    return 0
```

56. **end if**

When the function *TreeInclusion(D, Q)* is called in each recursion, the algorithm (line 1 - 7) will first make sure that D is large enough to "hold" Q, that is, it will compare their tree sizes to determine whether D is larger than or equal to Q. If the Q is a forest $(Q_1, \ldots, Q_q)$, the algorithm (line 2 - 3) will try to find the maximum number of subtrees of Q that D can include. Starting from the first subtree Q_1, it records the number of subtrees that D can include until it reaches a *critical point* i where $|(Q_1, \ldots, Q_i)| \leq |D| < |(Q_1, \ldots, Q_{i+1})|$. After that, the forest $(Q_1, \ldots, Q_i)$ with r_Q being a virtual root will be used (instead of Q) in the rest of the execution. The rest of algorithm flow (line 12 - 56) depends on the label test, which is to test if the label(r_D) is the same as the label(r_Q). This test is to satisfy the *label preservation condition* of the tree inclusion definition (see Section 1.3.2).

- If the test is true (since the virtual root matches anything, the forest $(Q_1, \ldots, Q_i)$ in the previous case will always pass this test), the algorithm (line 12 - 47) will use each of the subtrees of r_D: $D_1, \ldots, Dd$ to check against the subtrees of r_Q: $Q_1, \ldots, Q_q$ (or $Q_1, \ldots, Q_i$ if r_Q is a virtual root). This process is mainly performed in a *while* loop (line 22 - 34), where each loop is a recursive call *TreeInclusion(D_i, temp)* . In each recursive call, an integer x is returned to indicate how many subtrees of r_Q have been included. If $x = 0$, the algorithm (line 27 - 30) may need to check if D_i includes the first subtree of $r_{Q_{j+1}}$ since what it has done in each loop is to use the subtrees of r_{D_i} for checking, not D_i itself.

- If the test is false, i.e. two labels of r_D and r_Q are not the same, the algorithm (line 48 - 56) will try to find the first D_i that contains the whole Q by checking

the subtrees of r_D: $D_1, D_2, \ldots, D_d$ one by one against the Q.

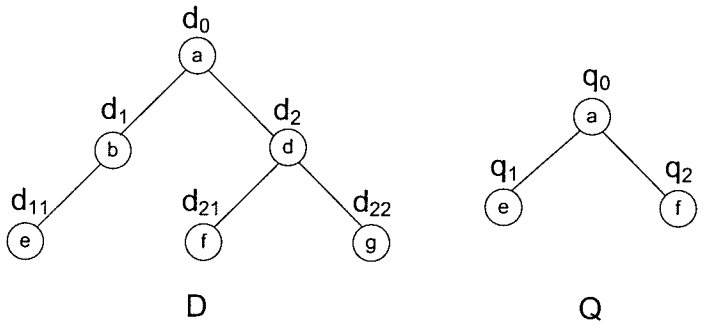
To better understand how this algorithm works, I give the following example to illustrate the algorithm logic. Suppose that we have a document tree D and a query tree Q as shown in Figure 3.4(a). Each node in D is identified as d_i, where $i = 0, 1, 2, \ldots$. And each node in Q is identified as q_i, such as q_0, q_1, q_2. We denote the subtree of a node as D_i or Q_i, where $i = 0, 1, 2, \ldots$. To find out whether D includes Q, we run Algorithm 1 as TreeInclusion(D, Q). The main steps are described as follows.

1. TreeInclusion(D, Q). Since label(d_0) = label(q_0) = "a" on line 12, we have a root-preserving embedding. First, i, j, x are set to their initial values: $i_0 = 1$; $j_0 = 0$; $x_0 = 0$. Then, we will use a *while* loop to check each subtree of d_0 against all subtree of q_0 until there is no more subtree of d_0 left or all the subtrees of q_0 are included. We first call TreeInclusion$(D_1, < Q_1, Q_2 >)$.

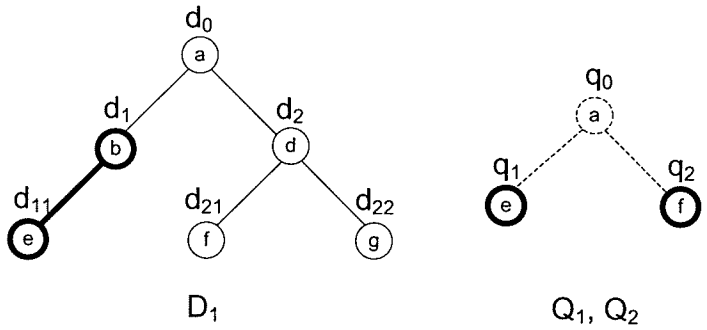
   (a) TreeInclusion$(D_1, < Q_1, Q_2 >)$. Since $< Q_1, Q_2 >$ is a forest (see Figure 3.4(b)), label(d_1) = label(virtual root) on line 12. $i_1 = 1$; $j_1 = 0$; $x_1 = 0$. Then, we will check D_{11} (subtree of d_1) against forest $< Q_1, Q_2 >$, i.e. calling TreeInclusion$(D_{11}, < Q_1, Q_2 >)$.

      i. TreeInclusion$(D_{11}, < Q_1, Q_2 >)$. Since $|D_{11}| < | < Q_1, Q_2 > |$, i.e. $< Q_1, Q_2 >$ is larger than D_{11}, we remove Q_2 from $< Q_1, Q_2 >$. Now, we check D_{11} against Q_1. Since label(d_{11}) = label(q_1) = "e", , d_{11} is a leaf node, and q_1 is not a virtual node, the algorithm returns 1 on line 14, which indicates that D_{11} includes Q_1.

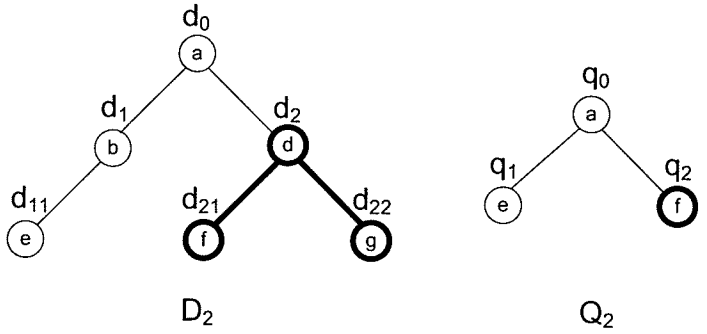
   After return from TreeInclusion$(D_{11}, < Q_1, Q_2 >)$ on line 23, $i_1 = 2$; $j_1 = 1$; $x_1 = 1$. We remove the included subtree Q_1 from $temp =< Q_1, Q_2 >$ on



(a) TreeInclusion (D, Q)



(b) TreeInclusion (D_1, <Q_1, Q2>)



(c) TreeInclusion (D_2, Q_2)

Figure 3.4: Illustration of the tree inclusion algorithm

line 25. Because there is no more subtree of d_1 left ($i_1 > 1$ on line 22), the *while* loop ends. However, since *temp* still has Q_2 and we also have a virtual root on $< Q_1, Q_2 >$, the algorithm returns $j_1 = 1$ on line 37.

After return from TreeInclusion($D_1, < Q_1, Q_2 >$) on line 23, $i_0 = 2$; $j_0 = 1$; $x_0 = 1$. We remove the included subtree Q_1 from *temp* $=< Q_1, Q_2 >$ on line 25. Because there is a subtree of d_2 left and Q_2 is not included yet, the *while* loop continues. We will check D_2 (subtree of d_2) against Q_2, i.e. calling TreeInclusion(D_2, Q_2).

2. TreeInclusion(D_2, Q_2) (see Figure 3.4(c)). Since label(d_2) $\neq$ label(q_2) on line 12, we have to begin with a *for* loop to check each subtree of d_2 against q_2 on line 49. We first call TreeInclusion(D_{21}, Q_2).

   (a) TreeInclusion(D_{21}, Q_2). Since label(d_{21}) = label(q_2) = "f", , d_{21} is a leaf node, and q_1 is not a virtual node, the algorithm returns 1 on line 14, which indicates that D_{21} includes Q_2.

   After return from TreeInclusion(D_{21}, Q_2) on line 50, there is no more subtree left since Q_2 has been included. the algorithm returns $x = 1$ on line 52.

3. The algorithm returns from TreeInclusion(D_2, Q_2) on line 23. After Q_2 is removed from *temp*, *temp* becomes empty. The *while* loop ends. Since the whole tree Q is included, the algorithm finally returns 1 on line 45 to indicate that D includes Q successfully.

3.1.3 Correctness of the Algorithm

In this section, we prove the correctness of the top-down tree inclusion algorithm.

Proposition 1. If Q is a tree, Algorithm 1 TreeInclusion(D, Q) returns 1 if D includes Q; otherwise it returns 0. If Q is a forest of the form: $< Q_1, \ldots, Q_j >$, TreeInclusion(D, Q) returns an integer i, indicating that D includes $< Q_1, \ldots, Q_i >$.

Proof. We prove the proposition by induction on the sum of the tree heights of D and Q, h. Without loss of generality, we assume that $height(D) \geq 1$ and $height(Q) \geq 1$.

When $h = 2$, we consider two cases.

1. Both D and Q are singular nodes: r_D and r_Q.

2. D is a singular node, but Q is a set of nodes.

In case 1, if r_D and r_Q have the same label, the algorithm returns 1 (see line 12); otherwise the algorithm returns 0 (see line 55). In case 2, a virtual root will be constructed for D, which matches any label. Then, the algorithm will check the subtrees of r_D against all the nodes in Q. Since r_D does not have any subtrees, the algorithm will return 0 (see line 16). Then, r_D will be checked against the first node in D and the algorithm returns 1 if they have the same label; otherwise, the algorithm returns 0 (see line 22-34).

Now, we make the induction hypothesis: when $h = n$, the proposition holds. We prove that the proposition still holds when $h = n+1$. Consider two trees D and Q with $height(D)+height(Q) = n+1$. Assume that Q is a tree. Let r_D and r_Q be the roots of D and Q, respectively. Let $D_1, \ldots, D_k$ be the subtrees of r_D, and let $Q_1, \ldots, Q_l$ be the subtrees of r_Q. Then, $height(D_i)+height(Q) \leq n$ and $height(D)+height(Q_j) \leq n$. If

$label(r_D) = label(r_Q)$, the algorithm partitions the integer sequence: $1, \ldots, l$ into some subsequences: $\{j_0{+}1, \ldots, j_1\}, \{j_1{+}1, \ldots, j_2\}, \ldots, \{j_{m-1}{+}1, \ldots, j_m\}$, where $j_0 = 0$ and $j_m \leq l$, such that each D_i ($i = 1, \ldots, m$; and $m \leq k$) includes $< Q_{j_{i-1}+1}, \ldots, Q_{j_i} >$ but not $< Q_{j_{i-1}+1}, \ldots, Q_{j_i}, Q_{j_i+1} >$ (see line 22-34). According to our hypothesis, the partition is correct. Thus, the algorithm will return 1 if $j_m = l$, indicating that D includes Q (see line 45); otherwise the algorithm returns 0 (see line 39). If $label(r_D) \neq label(r_Q)$, the algorithm will try to find the first D_i that includes the whole Q. According to the hypothesis, the return value must be correct. If Q is a forest in the form of $< Q_1, \ldots, Q_l >$, a virtual root will be constructed for it. In terms of the hypothesis, the algorithm will find the correct integer i such that D includes $< Q_1, \ldots, Q_i >$ (see line 36-37 and 42-43). It completes the proof.

3.2 Signature Techniques

Signature technique is the most important optimization technique used in our query evaluation method. The motivation of introducing signature technique into our method is based on two observations. First, since querying XML texts is usually much easier than querying XML structures, we can query XML documents directly based on their contents instead of their structures. Hence, if we immediately find out that a document will not match a query due to the differences between their contents, we will not bother checking their structures and simply skip it. Second, as we introduced in the Section 1.3.3, a signature is a binary bit pattern encoding a piece of text, multiple signatures can be superimposed together to get a new signature which provides a snapshot of all the texts represented by the multiple signatures, and the result of two signature comparison can tell us whether the two pieces of texts

represented by their corresponding signatures matches or not. Therefore, it is better for us to use signatures to handle document contents instead of directly manipulating texts.

3.3 Integration of Signatures and Top-down Tree Inclusion

To use signature technique, we must first generate signatures for all the XML documents that are expected to be searched for. For each of those XML documents, We use a hash function to generate a signature for each keyword in the text nodes. Since all the text nodes are leaves, which are at the bottom of the tree, we then superimpose all the signatures along the tree paths towards the root so that each node along the paths gets a superimposed signature. Eventually, the whole tree is decorated by the signatures, including a root signature representing all the texts in this document. For example, the tree in Figure 3.5(a) represents our sample document. We assign each keyword a signature. Figure 3.5(b) shows that when we superimpose these keyword signatures along the tree paths, the parents of the text nodes first get the signatures, which are the same signatures as their children's. Then, the node 'author' gets a signature by superimposing the signatures of its two children 'first' and 'last'. Finally, the root node 'book' gets its signature by superimposing the signatures of its four children 'title', 'author', 'publisher' and 'year'. The result of this process as shown in Figure 3.5(c) is that each tree node gets a signature with the same length. For a query tree, we use the same technique to generate signatures for it.

During a query evaluation, by each recursive call, the top-down tree inclusion

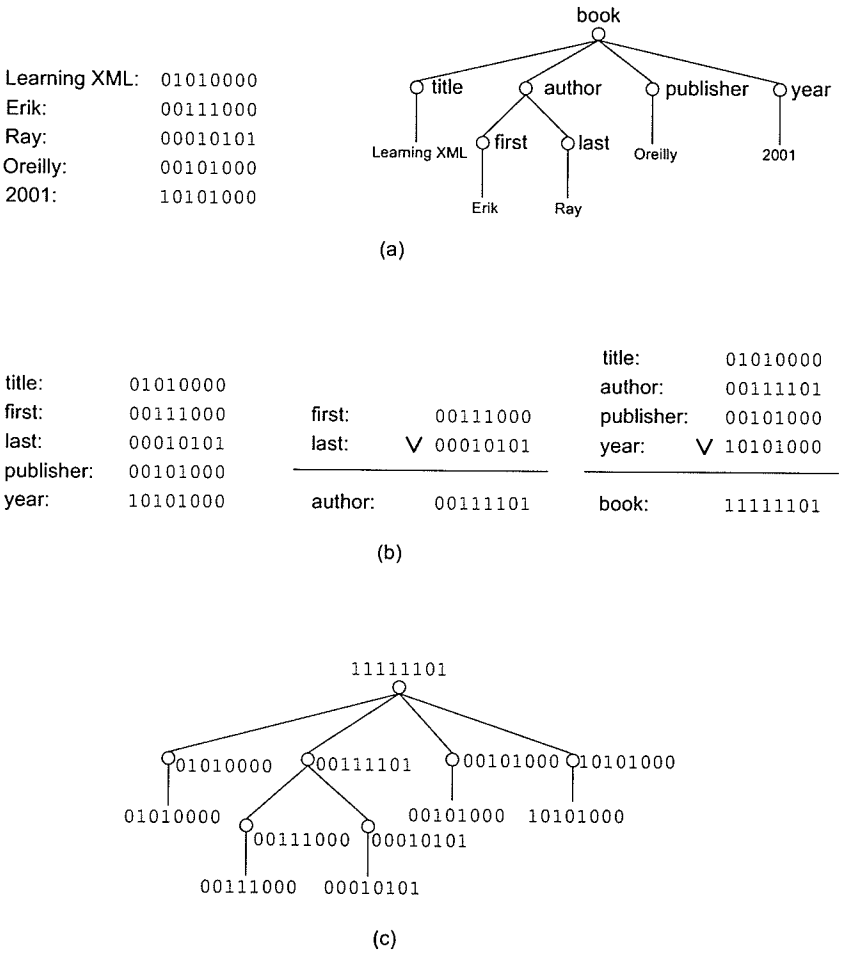


Figure 3.5: Signature a tree

algorithm compares the signatures of two tree nodes (in D and Q, respectively) prior to the comparison of two node labels. If the signature comparison fails, the algorithm knows that query keywords are absent in the current document subtree. Thus, this subtree will be cut off and the current recursive call quits immediately. For example, consider the trees T and S shown in Figure 3.6. We assign signatures to the nodes of T and S in the same way as the previous discussion. To check whether T includes S, we first compare the signatures of two root nodes, then compare their labels. Since both of them are the same, we will check whether the subtree of T that is rooted at node b includes the forest of S (i.e. two subtrees rooted at node c and node d, respectively). However, the signature of the node b in T does not match the signature of the virtual root a in S, that is, $s_b \wedge s_a \neq s_a$. Therefore, we do not need to further check the possible inclusion between these two trees. Our signature technique is especially useful when the initial comparison of two root signatures fails, since in that case the algorithm will skip the entire document. Imagine that when retrieving a large set of documents, our signature technique can save a lot of execution time, thus accelerate the whole evaluation process. To achieve such performance, we rely on a new signature technique called *Signature Tree*, which will be described in the next section.

3.4 Signature Tree Technique

To efficiently deal with large volume of data, such as hundreds of thousands of documents, we organize all document signatures into a *Signature Tree* [6], and use the signature tree to select the documents which may qualify for our query. This can save us a lot of time by directly jumping to those documents, which we really need

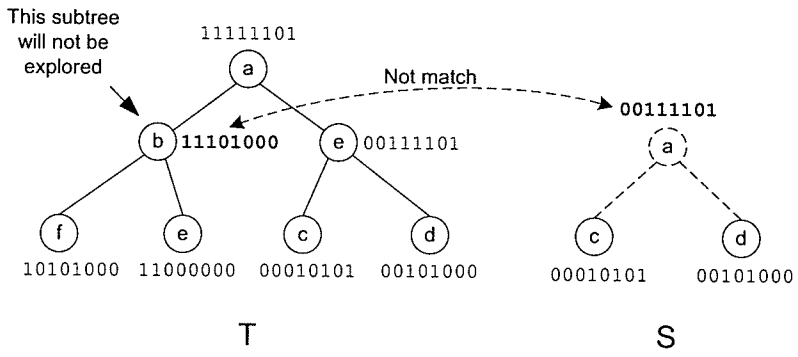


Figure 3.6: Accelerate evaluation by signatures

to check, rather than scanning all set of documents one by one.

3.4.1 Signature Tree

Like a signature file, a signature tree works for the same purpose, which is to let you submit a query signature and then find all the signatures matching this query signature. However, the way it works is much different from that of any traditional signature file, such as *BSSF* (Bit-Slice Signature Files) [4], *S-trees* [10], etc. As a new way of organizing signatures, a signature tree is a binary tree with each leaf node pointing to the corresponding signature in a signature file. Each of its internal node has three keys: the middle key (k_m) is associated with a number representing a bit position of the signatures; the left key (k_l) that points to the left child is always associated with 0; and the right key (k_r) that points to the right child is always associated with 1. The k_m tells which bit to be checked when doing a signature tree search. As an example, Figure 3.7 shows a signature tree representing a signature file

with eight signatures.

By organizing signatures into a binary tree, a signature tree can work in a way which is very similar to a binary search tree. Therefore, it can easily support searching and locating individual signatures from a very large set of signatures much faster than the traditional signature files can.

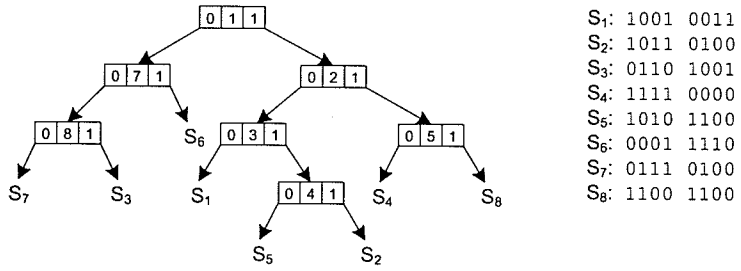


Figure 3.7: A sample of signature tree

3.4.2 Constructing a Signature Tree

If we do a search for signature s_5 or s_2 in the previous signature tree, it will take longer time than we search for other signatures. This is due to the fact that the tree is unbalanced. Therefore, an unbalanced signature tree could significantly increase the response time of signature queryings. To avoid constructing an unbalanced signature tree, we use a so-called *weight-based method* [6] to control the process of tree construction to produce a balanced or an almost balanced tree.

In the weight-based method, a signature file $F = \{s_1, s_2, \ldots, s_n\}$ is considered as a binary matrix in which each row is a signature and each column corresponds to a signature bit. We use $F[i]$ to denote the *ith* column of F. For each column $F[i]$, we

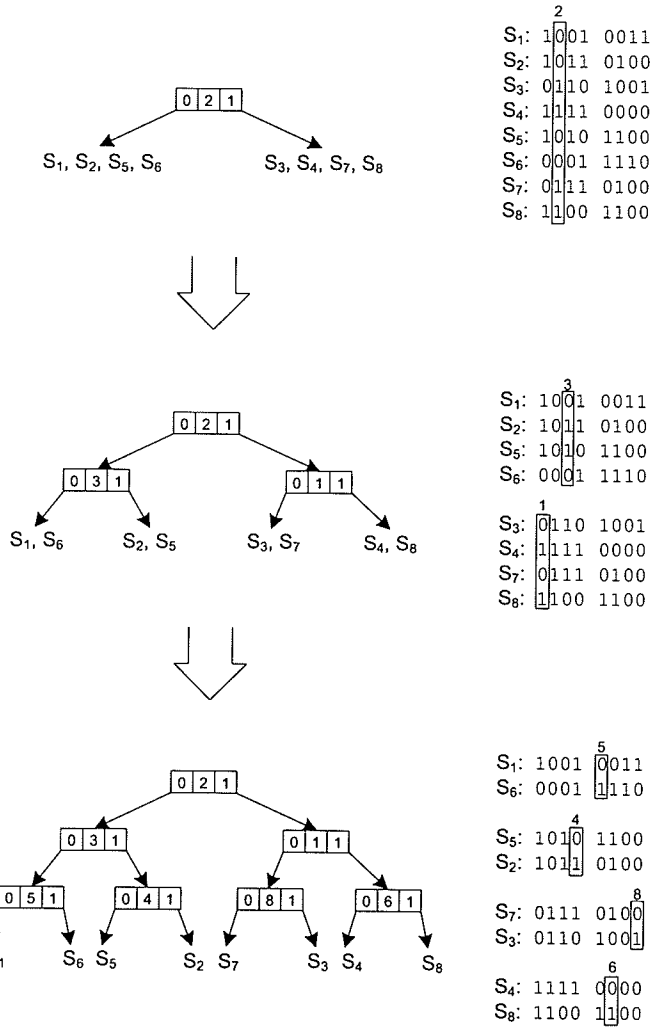


Figure 3.8: Constructing a balanced signature tree

compute its weight $w(F[i])$, which is the number of 1s appearing in the $F[i]$. To begin with constructing a signature tree, we arbitrarily select a column number c such that $w(F[c]) \simeq \frac{n}{2}$, i.e. $F[c]$ has almost the same number of 1 and 0. Then, we split F into two smaller signature files f_1 and f_2 based on c, which $f_1[c] = 0$ and $f_2[c] = 1$. Now, we can construct a simple tree T_c in which c is the middle key of the root node and the f_1, f_2 are the left and right child. Next, we treat each $f_i (i = 1, 2)$ as if it was the previous F, and repeat the same operation on f_1 and f_2, respectively. This generates two more trees from f_1 and f_2. Replacing the two child nodes f_1 and f_2 in the T_c with the corresponding trees, we "grow" the T_c with one more level. We continuously "grow" this tree until none of its leaf nodes can be replaced by a tree.

Algorithm 2 BalancedSigTree(F)

1. $N \leftarrow |F|$ //N gets the number of signatures in the F
2. **if** $N > 1$ **then**
3. choose c such that $w(F[c]) \simeq \frac{N}{2}$
4. Let $f_1 = s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ with $f_1[c] = 0$
5. Let $f_2 = s_{i_{k+1}}, s_{i_{k+2}}, \ldots, s_{i_N}$ with $f_2[c] = 1$
6. Create a node r
7. $r.k_m \leftarrow c$ //$r.k_m$ is the middle key of r
8. $r.k_l \leftarrow f_1$ //$r.k_l$ is the left key of r
9. $r.k_r \leftarrow f_2$ //$r.k_r$ is the right key of r
10. Replace the left child f_1 with BalancedSigTree(f_1)
11. Replace the right child f_2 with BalancedSigTree(f_2)
12. **end if**

Algorithm 2 is the formal description of the weight-based method. It is a recursive function which accepts a signature file as the input parameter and returns a balanced signature tree as the output. Figure 3.8 presents the steps of applying this algorithm to the signature file shown in the Figure 3.7. This time, a completely balanced signature tree is constructed.

1. Calling BalancedSigTree(F) with $F = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$. N first gets the value 8 on line 1. Then, on line 3, we choose $c = 2$ since $w(F[2]) = \frac{N}{2} = 4$. In this case, $f_1 = \{s_1, s_2, s_5, s_6\}$ and $f_2 = \{s_3, s_4, s_7, s_8\}$. From line 7 to 9, we create a root node with the middle key $k_m = 2$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11.

   (a) Calling BalancedSigTree(f_1) with $f_1 = \{s_1, s_2, s_5, s_6\}$. N first gets the value 4 on line 1. Then, on line 3, we choose $c = 3$ since $w(F[3]) = \frac{N}{2} = 2$. In this case, $f_1 = \{s_1, s_6\}$ and $f_2 = \{s_2, s_5\}$. From line 7 to 9, we create a root node with the middle key $k_m = 3$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11.

      i. Calling BalancedSigTree(f_1) with $f_1 = \{s_1, s_6\}$. N first gets the value 2 on line 1. Then, in line 3, we choose $c = 5$ since $w(F[5]) = \frac{N}{2} = 1$. In this case, $f_1 = \{s_1\}$ and $f_2 = \{s_6\}$. From line 7 to 9, we create a root node with the middle key $k_m = 5$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11. Since both f_1 and f_2 contain only a single signature, the recursive calls stop here.

      ii. Calling BalancedSigTree(f_2) with $f_2 = \{s_2, s_5\}$. N first gets the value 2 on line 1. Then, in line 3, we choose $c = 4$ since $w(F[4]) = \frac{N}{2} = 1$.

In this case, $f_1 = \{s_5\}$ and $f_2 = \{s_2\}$. From line 7 to 9, we create a root node with the middle key $K_m = 4$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11. Since both f_1 and f_2 contain only a single signature, the recursive calls stop here.

(b) Calling BalancedSigTree(f_2) with $f_2 = \{s_3, s_4, s_7, s_8\}$. N first gets the value 4 on line 1. Then, on line 3, we choose $c = 1$ since $w(F[1]) = \frac{N}{2} = 2$. In this case, $f_1 = \{s_3, s_7\}$ and $f_2 = \{s_4, s_8\}$. From line 7 to 9, we create a root node with the middle key $k_m = 1$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11.

i. Calling BalancedSigTree(f_1) with $f_1 = \{s_3, s_7\}$. N first gets the value 2 on line 1. Then, in line 3, we choose $c = 8$ since $w(F[8]) = \frac{N}{2} = 1$. In this case, $f_1 = \{s_7\}$ and $f_2 = \{s_3\}$. From line 7 to 9, we create a root node with the middle key $k_m = 8$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11. Since both f_1 and f_2 contain only a single signature, the recursive calls stop here.

ii. Calling BalancedSigTree(f_2) with $f_2 = \{s_4, s_8\}$. N first gets the value 2 on line 1. Then, in line 3, we choose $c = 6$ since $w(F[6]) = \frac{N}{2} = 1$. In this case, $f_1 = \{s_4\}$ and $f_2 = \{s_8\}$. From line 7 to 9, we create a

root node with the middle key $k_m = 6$, the left key k_l pointing to f_1 and the right key k_r pointing to f_2. We replace f_1 with a recursive call BalancedSigTree(f_1) and f_2 with a recursive call BalancedSigTree(f_2) on line 10, 11. Since both f_1 and f_2 contain only a single signature, the recursive calls stop here.

3.4.3 Querying a Signature Tree

Once a balanced signature tree is constructed from a set of document signatures, we can query the signature tree with a query signature s_q to find all the document signatures matching s_q, and retrieve only the corresponding documents. By matching, we mean that the result of bitwise ORing between any document signature and the query signature is the query signature itself. As mentioned before, a signature tree works in a manner analogous to a binary search tree. Suppose that we have a query signature s_q. The *ith* bit position of s_q is denoted as $s_q(i)$. To find all the match signatures in the signature tree, we start from the root and walk down the tree in a depth-first manner. Let v be the node that we encountered each time, we will check $s_q(k_m)$ as follows to determine which nodes will be explored in a next walk.

- If $s_q(k_m) = 1$, we will only go to the right child of v.
- If $s_q(k_m) = 0$, we will explore both children of v.

The searching will stop until we reach some leaf nodes. Then, the signatures that are pointed by these leaf nodes will be verified with s_q to see if they are the real matches or not. Algorithm 3 implements the above querying logic. When we call the algorithm with the root of a signature tree and a query signature, it will find all

Algorithm 3 QuerySigTree(v, s_q)

```
1.  R ← φ      //R is the result set of matching signatures, is empty initially.
2.  S ← v      //S is a stack.
3.  while S is not empty do
4.    v ← S
5.    if v is a leaf node and s_v ∧ s_q = s_q then
6.      R ← R ∪ s_v      //s_v is the signature pointed by v.
7.    else
8.      i ← v.k_m      //v.k_m is the middle key of v.
9.      if s_q(i) = 1 then
10.       S ← v_r      //v_r is the right child of v.
11.     else
12.       S ← v_r
13.       S ← v_l      //v_l is the left child of v.
14.     end if
15.   end if
16. end while
```

matching document signatures for us. For example, in Figure 3.9, we have a query signature $s_q = 01001001$, and we are going to query the same balanced signature tree shown in Figure 3.8. We denote a tree node as v_{k_m}, where k_m is the middle key of the node.

1. Calling QuerySigTree(v_2, s_q) with v_2 being the root of the signature tree. Initially, the result set R is empty. v_2 is pushed into a stack S on line 2. Since the stack S is not empty, we begin with a while loop. v_2 pop up from the S on line 4. Since v_2 is not a leaf node, We go to line 8 and set the bit position $i = 2$ (the middle key of v_2). Then, we check the $2nd$ position of s_q and find out $s_q(2) = 1$, thus, v_1 (i.e. the right child of v_2) is pushed into the S on line 10.

2. Since $S = (v_1)$, we begin with a new loop. v_1 pop up from the S on line 4. Since v_1 is not a leaf node, We go to line 8 and set the bit position $i = 1$ (the

middle key of v_1). Then, we check the $1st$ position of s_q and find out $s_q(1) = 0$, thus, v_6 and v_8 (i.e. both children of v_1) are pushed into the S on line 10.

3. Since $S = (v_6, v_8)$, we begin with a new loop. v_8 pop up from the S on line 4. Since v_8 is not a leaf node, we go to line 8 and set the bit position $i = 8$ (the middle key of v_8). Then, we check the $8th$ position of s_q and find out $s_q(8) = 1$, thus, the right child of v_8) are pushed into the S on line 10.

4. Since $S = (v_6$, the right child of $v_8)$, we begin with a new loop. The right child of v_8 pop up from the S in line 4. Since it is a leaf node and $s_3 \wedge s_q = s_q$ (i.e. s_3 is a real match), $R = \{s_3\}$.

5. Since $S = (v_6)$, we begin with a new loop. v_6 pop up from the S on line 4. Since v_6 is not a leaf node, we go to line 8 and set the bit position $i = 6$ (the middle key of v_6). Then, we check the $6th$ position of s_q and find out $s_q(6) = 0$, thus, both children of v_6 are pushed into the S on line 10.

6. Since $S =$ (the right child of v_6, the left child of v_6), we begin with a new loop. The left child of v_6 pop up from the S on line 4. Since it is a leaf node but $s_4 \wedge s_q \neq s_q$, R remains $\{s_3\}$.

7. Since $S =$ (the right child of v_6), we begin with a new loop. The right child of v_6 pop up from the S on line 4. Since it is a leaf node but $s_8 \wedge s_q \neq s_q$, R remains $\{s_3\}$.

8. Now S is empty, the while loop ends. At last, we find that s_3 matches our query signature s_q.

From this example, we saw the benefit of the signature tree. Instead of sequentially scanning all signatures, we only need to check 3 signatures. This is why querying a signature tree will be more efficient than other signature file methods in the case of dealing with a huge amount of signatures.

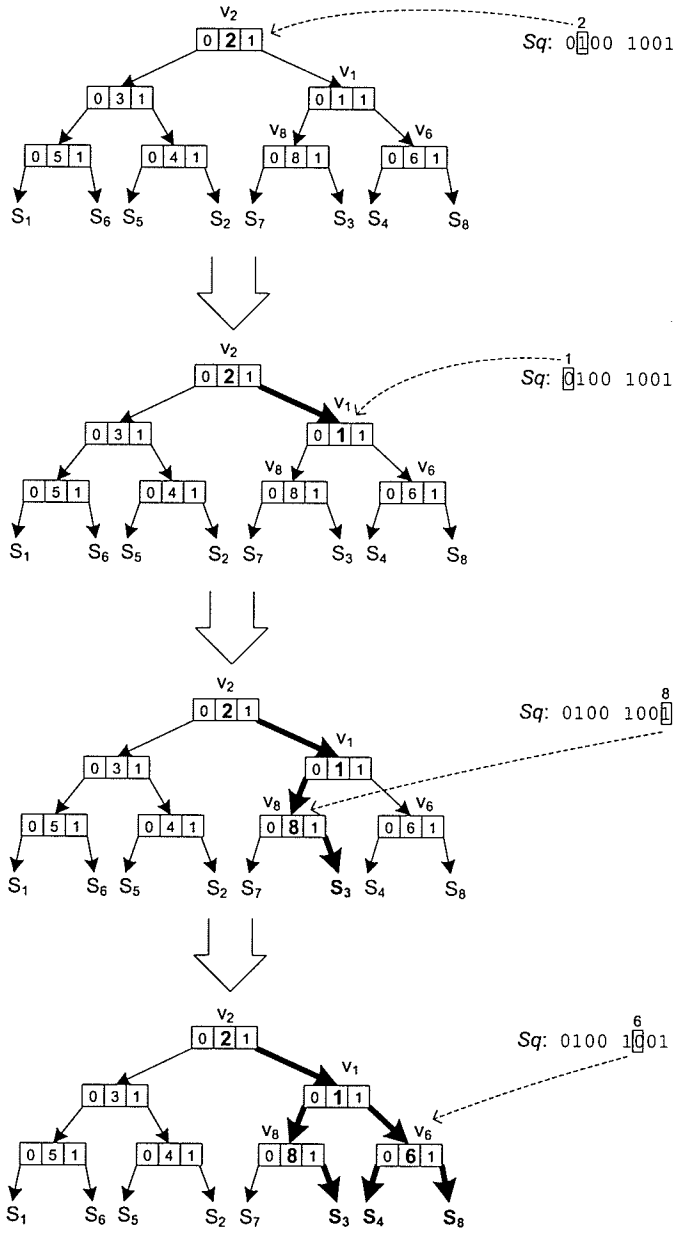


Figure 3.9: Querying a balanced signature tree

Chapter 4

Performance Evaluation

In this chapter, we first describe some implementation issues that we encountered when we were conducting the experiments to study the performance of the path-oriented query evaluation method proposed in this thesis. Then, we present a performance study of this method. We will discuss the details of how we performed the experiments, and analyze the experimental results.

4.1 Implementation

In order to study the behavior of our propose method, we implemented our path-oriented query evaluation method and carried out a series of performance experiments. During the actual implementation, we had to face two challenges. The first challenge was to determine what signature length we used in our signature technique. The second challenge was to find a way to store XML documents into a relational database. In this section, we address these two issues.

4.1.1 Determining the Length of Signatures

When we were implementing the signature technique, an important problem we encountered is what signature length should be applied for all the signatures . The Formula 1.1, which we discussed in the background section, is not useful any more for our problem. Due to the superimposing signatures along the tree paths, each subtree is considered as a block. Since the number of words in each subtree is not equally distributed, the requirement that each block contains the equal number of words is no longer satisfied. Thus, we can not use that formula to calculate the signature length. For this reason, we make the following analysis and develop a new way to estimate the signature length so that the previous problem can be removed.

Given two signatures s_1, s_2, both of them are of length F and with m_1 and m_2 bits set to 1, respectively. If we superimpose them (i.e. $s = s_1 \vee s_2$) to get a new signature s, s will possibly have more bits set to 1 than either of them. To keep the ratio of 1 bits (i.e. the number of bits set to 1 versus the signature length) in s unchanged, s has to be extended longer. The question is how long it should be. To answer this question, we observe that for each bit position at which s_1 and s_2 have different bit value, the length of s is increased by 1. Hence, our objective is to estimate how many bits the two signatures will not be the same. In the worst case, the s_1 and s_2 do not have any bit position in common, thus, the length of s will be $m_1 + m_2$ bits. However, in general case, the s_1 and s_2 will have some bit positions in common. Let λ be a random variable representing the number of such positions. The following mathematical expectation of λ is the number of bit positions at which

both s_1 and s_2 set 1.

$$\varepsilon(\lambda) = \sum_{\lambda=1}^{m_{min}} (\lambda \times p(\lambda)) \tag{4.1}$$

where $m_{min} = min(m_1, m_2)$, $p(\lambda)$ represents the probability that s_1 and s_2 have λ bits in common, and $1 \leq \lambda \leq m_{min}$. $p(\lambda)$ can be calculated as follows:

$$p(\lambda) = \frac{\binom{F-m_2}{m_1-\lambda}\binom{m_2}{\lambda}}{\binom{F}{m_1}} \tag{4.2}$$

Now, the number of bit positions $\varepsilon(m_\delta)$, where s_1 and s_2 are not the same, can be calculated as follows:

$$\varepsilon(m_\delta) = m_{max} - \varepsilon(\lambda) \tag{4.3}$$

where $m_{max} = max(m_1, m_2)$. We notice that $\varepsilon(m_\delta)$ is in fact the amount by which we have to increase the length of s when superimposing s_1 and s_2. Therefore, the length of s will be $F + \varepsilon(m_\delta)$. Based on the above formulas, we can easily compute the length of signatures for a document tree. Suppose that an XML document has n keywords. We first calculate the average number of keywords per text node, then use it as the value of D in Formula 1.1 to compute an initial value of F. This value will be increased when we superimpose n signatures $s_1, s_2, \cdots, s_n$ together. To determine the final signature length after superimposing the n signatures, we first calculate m_δ for superimposing s_1 and s_2 using Formula 4.3 to get a new length F_{12}, which represents the length of signature $s_{12} = s_1 \vee s_2$. We then calculate a new length F_{13} for signature $s_{13} = s_{12} \vee s_3$ based on the previous length F_{12}. This computation is stopped until we get a length F_{1n} for signature $s_{1n} = s_{1(n-1)} \vee s_n$. The length F_{1n} becomes our final signature length used for all signatures. To get a sense of how the above formula performs in terms of different initial parameters, we conducted a test with three set of parameters: $x/y = (3/6, 4/8, 5/10)$, where x represents how many bits will be set

to 1, and y represents the initial signature length. The test result is presented in Figure 4.1.

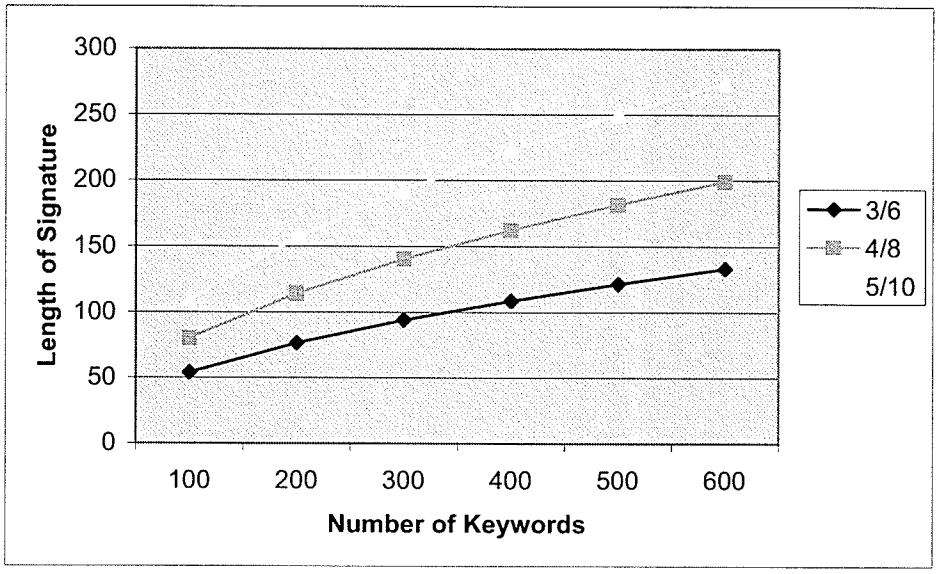


Figure 4.1: The test of new signature formula

4.1.2 Mapping XML data into a Relational Database

In this thesis, we focus our attention on strategies to query XML documents from relational databases. Therefore, before we can make any query evaluation on XML documents, we need to load all test documents into a relational database, i.e. we must define a mapping scheme that can map XML data into a relational database.

Our goal is to select a mapping scheme that will retain the tree structures of XML documents after we loaded the documents into a database so that the top-down tree inclusion algorithm can take advantages of these tree structures. The mapping scheme that we chose consists of the following three relations (the primary keys are underlined):

- **Element** (docID, tagID, tagName, firstChildID, siblingID, textID, treeSize, signature, attributeID)
- **Text** (docID, textID, textValue)
- **Attribute** (docID, attributeID, tagID, attributeName, attributeValue)

In the Element relation, we defined the following fields:

docID represents the identifier of the document that the element node belongs to.

tagID represents the identifier of this element.

tagName is the name of the element (or tag).

firstChildID is the pointer to the first child of the element node.

siblingID is the pointer to the right sibling of the element node.

attributeID is the pointer to the first attribute in the element.

textID is the identifier of the text node if the element node has a child which is a text node.

treeSize is the size of the tree whose root is the element node;

signature is the signature (in a decimal format) associated with the current element.

In the Text relation, we defined the following fields:

docID See above.

textID See above.

textValue is the content of the text node.

In the Attribute relation, we defined the following fields:

docID See above.

attributeID is the ID of the attribute (including the first attribute as above).

tagID See above.

attributeName is the name of the attribute.

attributeValue is the value of the attribute.

The mapping operation is done by a depth-first traversal of each document tree. For each node in the document tree, we determine its type and act accordingly. Each node in the document tree is traversed only once. Thus, each record in the Element or Text relation corresponds to a tree node in the document tree. Take the document in Figure 1.1 as an example, it will be mapped to two relations (since there is no attribute in this document, the Attribute relation table will be empty). Table 4.1 is the Element relation of the document, and table 4.2 is the Text relation of the document.

Table 4.1: A sample of the Element relation

docID	tagID	tagName	firstChildID	siblingID	textID	treeSize	...
1	1	book	2			7	
1	2	title		3	1	1	
1	3	author	4	6		3	
1	4	first		5	2	1	
1	5	last			3	1	
1	6	publisher		7	4	1	
1	7	year			5	1	

Table 4.2: A sample of the Text relation

docID	textID	textValue
1	1	Learing XML
1	2	Erik
1	3	Ray
1	4	Oreilly
1	5	2001

4.2 Experimental Setup

We conducted our experiments on a Pentium IV 1.8 GHz PC with 512 MB RAM and 30 GB hard disk, running Windows 2000 Professional with Service Pack 4. We chose Oracle9i Database Release 2 Enterprise Edition for Windows as the RDBMS platform. All buffer caches of Oracle database are set to use default sizes. To avoid the impact from the network latencies and communication overheads between client and server, we developed all algorithms in Oracle PL/SQL and Java, and stored them as Oracle stored procedures inside the database. For mapping XML documents into the Oracle database, we used Oracle XML Developer's Kit (XDK for PL/SQL version 9.2) to parse and process XML documents.

4.3 Data Sets

Our experiments are based on two real data sets: Shakespeare's plays in XML [2] and Digital Bibliography and Library Project database (DBLP) [16]. They are free and easy to get from the Internet.

- The Shakespeare's plays in XML is a collection of XML documents, each of which is the script of a Shakespeare's play represented in XML format. The

data set consists of 37 documents with total file size 8 MB. Basically, all the documents share a very similar tree structure which is shown in Figure 4.2. Each document tree is quite large, approximately containing 8900 tree nodes (4900 element nodes and 4000 text nodes). The average depth of the tree is 6.

- The Digital Bibliography and Library Project database (DBLP) is the popular computer science bibliography in XML format. It includes conference papers, articles, etc. The original data set is one huge file with file size 150 MB. For the experimental purpose, we split the file into many smaller XML documents, each of which is a corresponding bibliographic record of a publication. The new data set consists of 300000 XML documents. Each document of DBLP is much smaller than that of the Shakespeare data set, which averagely has 10 element nodes and 10 text nodes.

Some quantitative characteristics of the data sets are summarized in Table 4.3. As we can see, the two data sets covers a wide range of tree sizes and depths. To study the impacts of the variations of query patterns and their matching positions on performance, we use Shakespeare data set. To explore the scalabilities of different methods, we use DBLP.

Table 4.3: The characteristics of data sets

	Shakespeare	**DBLP**
Number of documents	37	300,000
Size of data set (MB)	8	150
Total number of elements	179,689	3,140,287
Total number of texts	147,442	3,059,052
Number of elements per document	4,856	10
Number of texts per document	3,985	10
Average tree depth	6	2

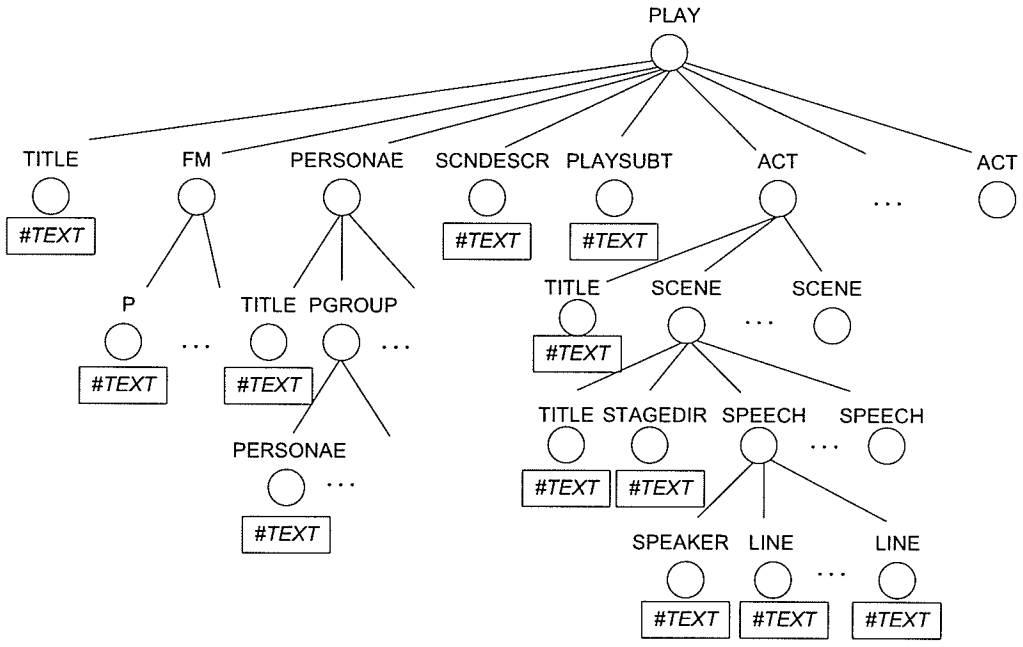


Figure 4.2: The tree structure of a Shakespeare's play

4.4 Comparable Methods

We implemented and experimented with the following four methods of path-oriented query evaluation to study and compare their performances.

- **IEW** (Inverted index on Elements and Words): the method [23] which maps XML elements and words (see Section 2.1) into two relations (E-index and T-index relations) and processes containment queries in the RDBMS.
- **IPW** (Inverted index on Paths and Words): the method [21] which maps four inverted indexes (see Section 2.1) into four relations (Path, PathIndex, Term, and TermIndex relations) and processes containment queries in the RDBMS.
- **ViST**: the method [22] which uses the subsequence (non-contiguous) matching of two structure-encoded sequences to answer XML queries without involving any join operations.
- **TIS** (Tree Inclusions and Signatures): our method which integrates the top-down tree inclusion algorithm with the signature technique.

4.5 Requirements of Data Storage

The storage requirement is one of the performance metrics we were going to measure. Since each method has its own schema of data storage, we loaded two data sets into the Oracle database separately for each method that we are going to test. The actual spaces used by each method are listed in Table 4.4 and Table 4.7. Table 4.8 compares the total sizes of relational tables that are required for these methods. From the

Table 4.4: Table sizes (MB) of IEW method

	Shakespeare	DBLP
E-index	8	160
T-index	10	240

Table 4.5: Table sizes (MB) of IPW method

	Shakespeare	DBLP
Term	5	80
TermIndex	19	240
Path	< 1	< 1
PathIndex	6	98

Table 4.6: Table sizes (MB) of ViST method

	Shakespeare	DBLP
D-Ancestorship	22	380
S-Ancestorship	8	120

Table 4.7: Table sizes (MB) of TIS method

	Shakespeare	DBLP
Elements	12	250
Texts	8	120

Table 4.8: Comparison of total table sizes (MB)

	Shakespeare	DBLP
IEW	18	400
IPW	20	420
ViST	30	500
TIS	20	370

comparison, we see that TIS method uses the least space to store DBLP data set among the four methods.

4.6 Experiment on Shakespeare data set

In this experiment, we performed tests on the Shakespeare data set. As we know, XML queries have a variety of patterns, but they are usually much smaller than documents in terms of sizes (the number of tree nodes), and they can be embedded in the different parts of the documents. Shakespeare data set has only 37 documents, but each of those documents actually is a very big and complex tree. Because of this characteristic of the Shakespeare data set, it is suitable for us to study the impacts of the variations of query patterns and their matching positions on performance.

4.6.1 Queries

We tested 25 queries which are organized into 5 groups as shown in Tables 4.9 - 4.13. The syntax of path expressions is borrowed from XPath [9], and is simplified for the sake of easy understanding. '/' represents a parent-child relationship, '//' represents an ancestor-descendant relationship. The expressions inside a pair of square brackets are predicates. '|' connects different paths together.

Table 4.9: Group I. Queries with incremental path lengthes

Query	Path Expression
Q1	/PLAY//'magnificence'
Q2	/PLAY/ACT//'magnificence'
Q3	/PLAY/ACT/SCENE//'magnificence'
Q4	/PLAY/ACT/SCENE/SPEECH//'magnificence'
Q5	/PLAY/ACT/SCENE/SPEECH/LINE/'magnificence'

Table 4.10: Group II. Queries with incremental degrees

Query	Path Expression
Q6	/PLAY//LINE/'magnificence'
Q7	/PLAY//[LINE/'magnificence'|LINE/'churchyard']
Q8	/PLAY//[LINE/'magnificence'|LINE/'churchyard'| LINE/'reverence']
Q9	/PLAY//[LINE/'magnificence'|LINE/'churchyard'| LINE/'reverence'|LINE/'frequent']
Q10	/PLAY//[LINE/'magnificence'|LINE/'churchyard'| LINE/'reverence'|LINE/'frequent'|LINE/'heirless']

Table 4.11: Group III. Queries matching at higher level of the document

Query	Path Expression
Q11	/PLAY//[LINE/'magnificence'|LINE/'perpetuity']
Q12	/PLAY//[LINE/'churchyard'|LINE/'ladyship']
Q13	/PLAY//[LINE/'reverence'|LINE/'continent']
Q14	/PLAY//[LINE/'frequent'|LINE/'linen']
Q15	/PLAY//[LINE/'heirless'|LINE/'delivery']

Table 4.12: Group IV. Queries matching at middle level of the document

Query	Path Expression
Q16	/SCENE//[LINE/'magnificence'|LINE/'utterance']
Q17	/SCENE//[LINE/'churchyard'|LINE/'barbarism']
Q18	/SCENE//[LINE/'reverence'|LINE/'carriage']
Q19	/SCENE//[LINE/'frequent'|LINE/'imagination']
Q20	/SCENE//[LINE/'heirless'|LINE/'successor']

Table 4.13: Group V. Queries matching at lower level of the document

Query	Path Expression
Q21	/SPEECH//[LINE/'magnificence'|LINE/'unintelligence']
Q22	/SPEECH//[LINE/'churchyard'|LINE/'crickets']
Q23	/SPEECH//[LINE/'reverence'|LINE/'ceremonious']
Q24	/SPEECH//[LINE/'frequent'|LINE/'exercise']
Q25	/SPEECH//[LINE/'heirless'|LINE/'companion']

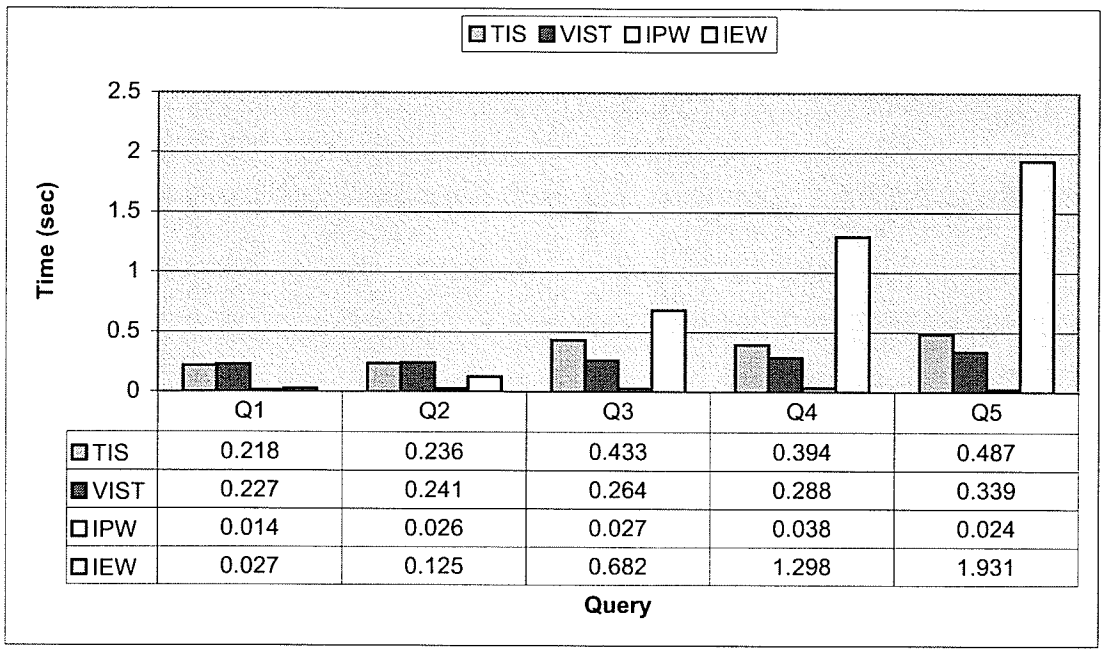
The queries in Group I is to test the impact of path lengths on performance. The queries in Group II is to test the impact of node degrees on performance. The queries in Group III - V are to test the impact on performance when query trees are embedded in different parts of a document, and in the same group, the queries are embedded in the same subtree level and follow the left-to-right order.

4.6.2 Test Method

Due to a relational database like Oracle that always has some sort of cache systems such as a buffer pool to optimize the response time of every submitted query, we did not test queries in a consecutive manner, instead, we shut down and restarted the database after each test. In such a way, we avoided the situation that the execution time for the same query would be shorter and shorter if we are running this query repeatedly. We run through each group five times, and recorded an average execution time for each query as the final test result.

4.6.3 Results

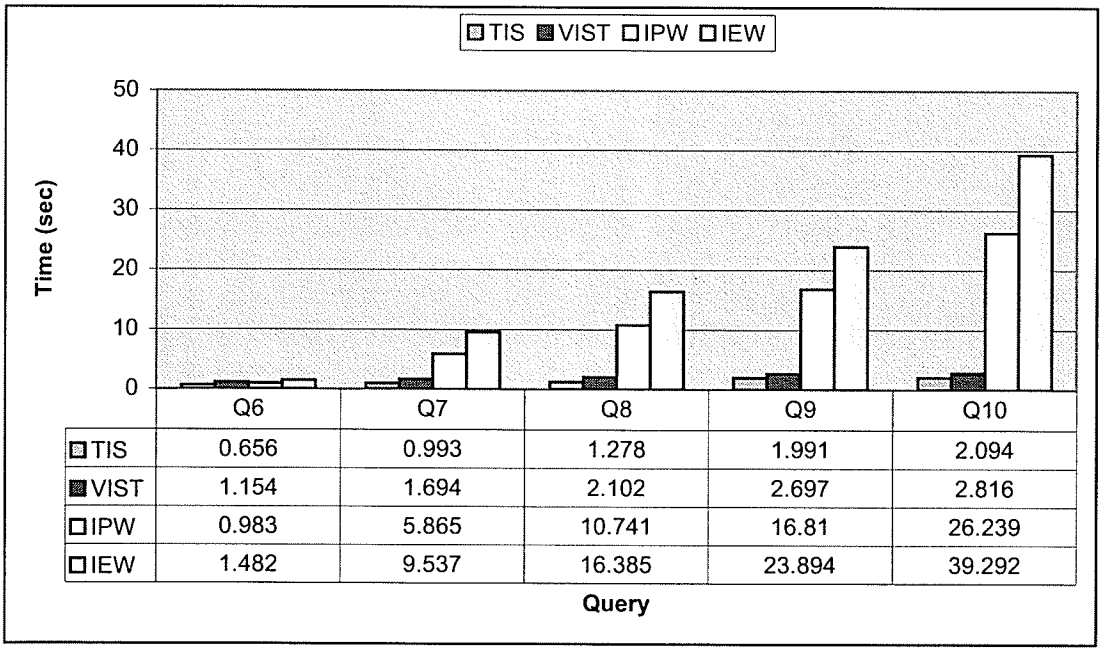
Figure 4.3 demonstrates the test results of Group One. From the figure, we can see that the TIS method is not as efficient as the IPW method in these tests, but it is comparable to the ViST method. Because of the queries with only one single path involved, the IPW method performs only two joins: 1) the join between relation Path and WordIndex; 2) the join between relation Word and the result of the first join. Although the TIS method needs only scan a single path once, it has to perform two operations during the scan of a query sequence: one is for the label checking and the other is for finding the first child or the direct right sibling of the corresponding node.



	Q1	Q2	Q3	Q4	Q5
TIS	0.218	0.236	0.433	0.394	0.487
VIST	0.227	0.241	0.264	0.288	0.339
IPW	0.014	0.026	0.027	0.038	0.024
IEW	0.027	0.125	0.682	1.298	1.931

Query

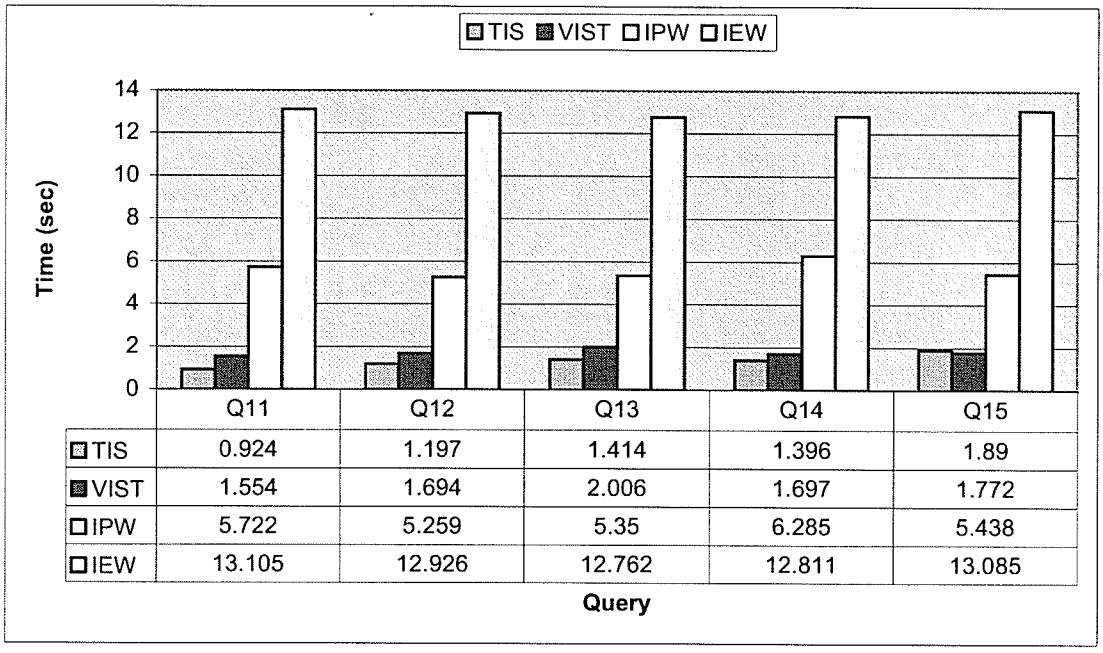
Figure 4.3: Execution time of queries in Group One



	Q6	Q7	Q8	Q9	Q10
TIS	0.656	0.993	1.278	1.991	2.094
VIST	1.154	1.694	2.102	2.697	2.816
IPW	0.983	5.865	10.741	16.81	26.239
IEW	1.482	9.537	16.385	23.894	39.292

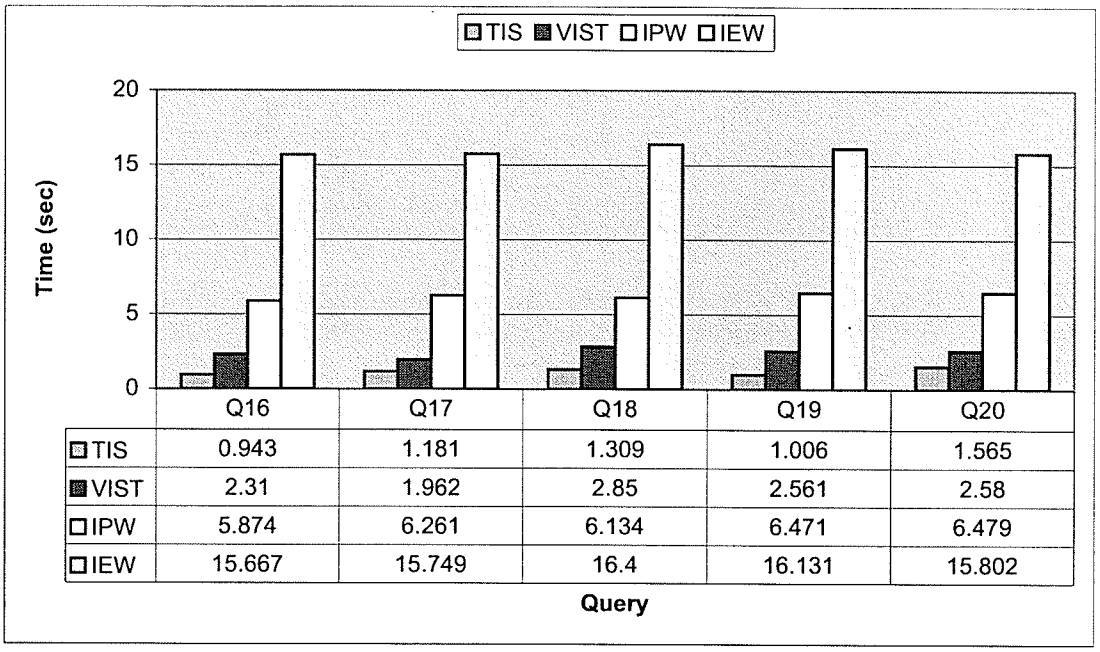
Query

Figure 4.4: Execution time of queries in Group Two



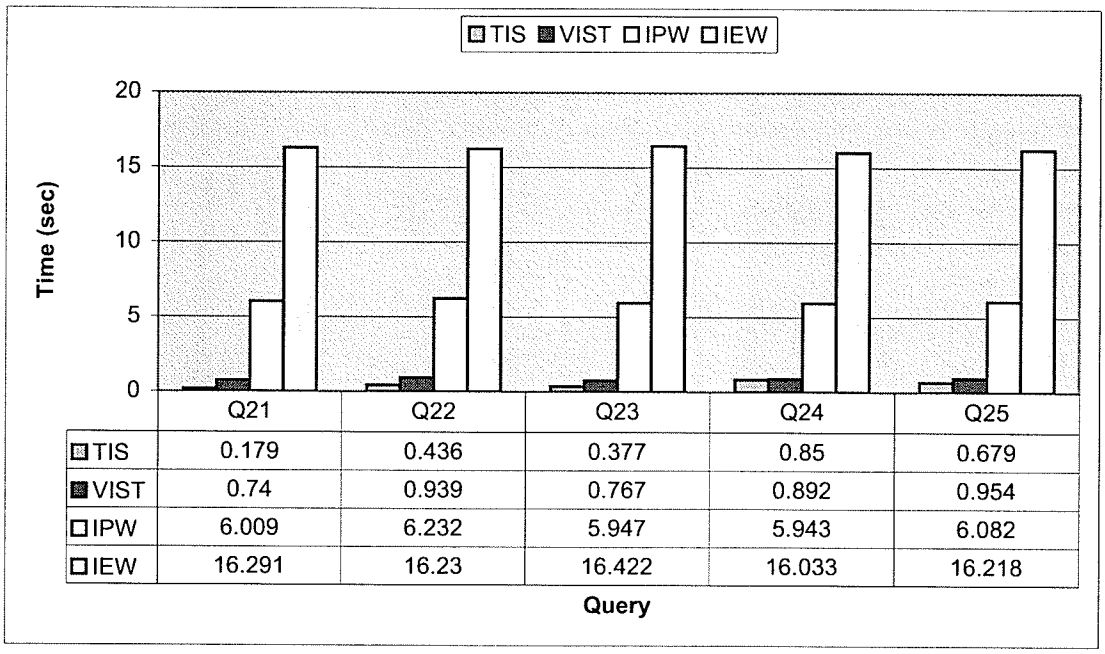
	Q11	Q12	Q13	Q14	Q15
TIS	0.924	1.197	1.414	1.396	1.89
VIST	1.554	1.694	2.006	1.697	1.772
IPW	5.722	5.259	5.35	6.285	5.438
IEW	13.105	12.926	12.762	12.811	13.085

Figure 4.5: Execution time of queries in Group Three



	Q16	Q17	Q18	Q19	Q20
TIS	0.943	1.181	1.309	1.006	1.565
VIST	2.31	1.962	2.85	2.561	2.58
IPW	5.874	6.261	6.134	6.471	6.479
IEW	15.667	15.749	16.4	16.131	15.802

Figure 4.6: Execution time of queries in Group Four



	Q21	Q22	Q23	Q24	Q25
TIS	0.179	0.436	0.377	0.85	0.679
VIST	0.74	0.939	0.767	0.892	0.954
IPW	6.009	6.232	5.947	5.943	6.082
IEW	16.291	16.23	16.422	16.033	16.218

Figure 4.7: Execution time of queries in Group Five

Therefore, as for single path queries, the TIS method is always less efficient than the IPW method. For short path queries (e.g. Q1, Q2), the IEW method works much better than TIS and ViST, however, the longer a path query is, the more inefficient the IEW method becomes.

The results from Group Two are shown in Figure 4.4. As more and more paths are involved in queries (from Q6 to Q10), the execution time of both IPW and IEW increased dramatically. To answer each query, both methods have to decompose the query into multiple single paths, and evaluate each path separately, then combine the results of single path queries. Since each combination has to check the common ancestors of different paths, this is a very time-consuming task. In contrast, both TIS and ViST performed much better than the IPW and IEW methods did.

The results from the rest three groups (Figure 4.5, 4.6 and 4.7) are similar to

each other no matter where a matching takes place. The IEW method did much worse than other three methods in this three tests, it spent 13 to 16 seconds each to answer most queries, whereas the TIS method outperformed other methods with around 1 second for answering each query.

The results of this experiment suggest us that both IEW and IPW are inefficient in answering multi-path queries due to many join operations, as we expected. When the queries became more complex (i.e. more nodes and paths were involved in the queries), the ViST method was inefficient too, it needed more execution time than the TIS method did since it had to recursively issue a range query to find all the descendants of the current node that satisfied S-Ancestorships when scanning each node of a query sequence. This type of operations appeared to be costly. Contrary, the signature technique made the TIS method skip many query nodes and paths that were unnecessary for it to check during executions.

4.7 Experiment on DBLP data set

In this experiment, we performed tests on the DBLP data set. As we mentioned before, DBLP is a huge data set that has total 300000 XML documents. Hence, it is very suitable for us to study the scalabilities of the previous four methods (IEW, IPW, ViST and TIS).

4.7.1 Queries

To study the scalabilities of the methods, we organized test queries into 3 groups according to query sizes: small, median and large. Each group has 5 queries, which

are DBLP documents in practice since the documents in the DBLP data set are small enough to be considered as queries. As the previous experiments, we run tests on each query 5 times to get an average execution time as a final result. Between two consecutive tests, we always shut down the database and restarted the database again.

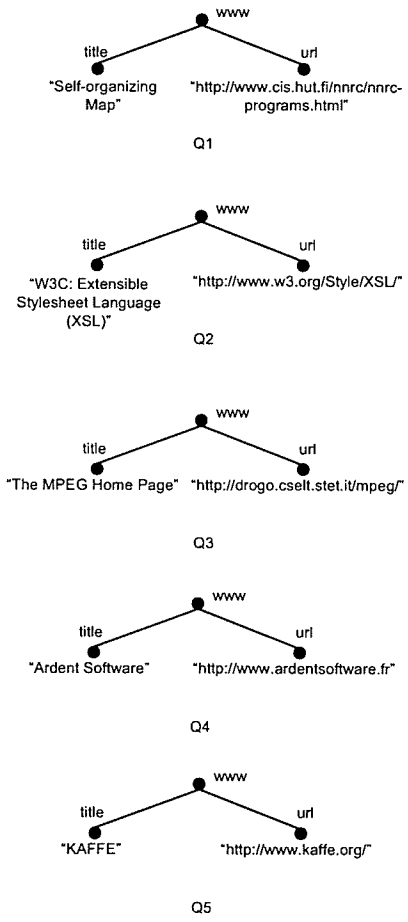


Figure 4.8: Queries of small sizes

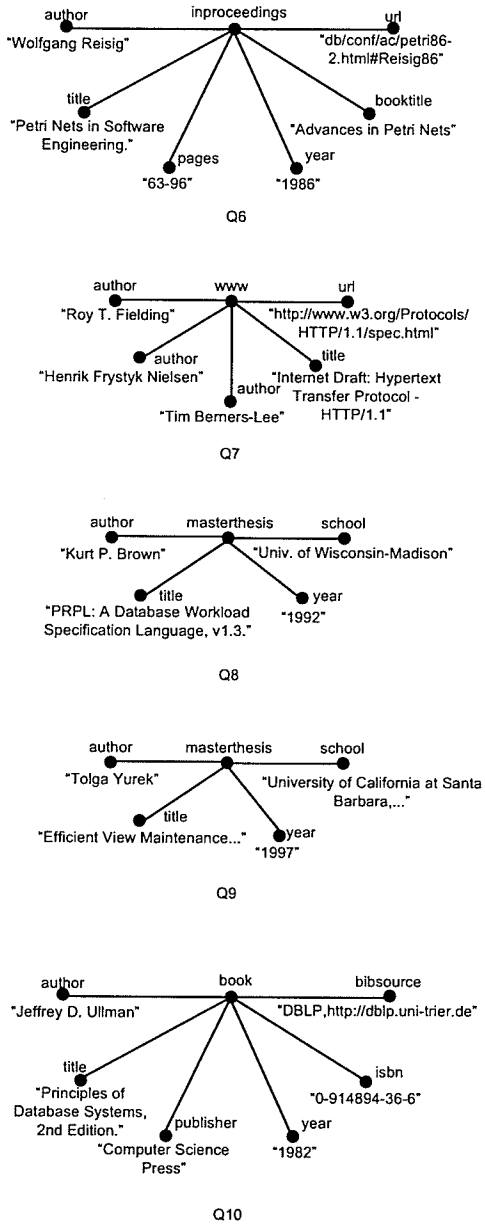
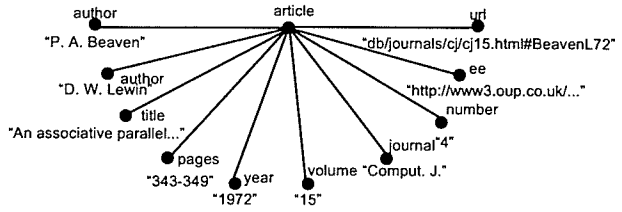
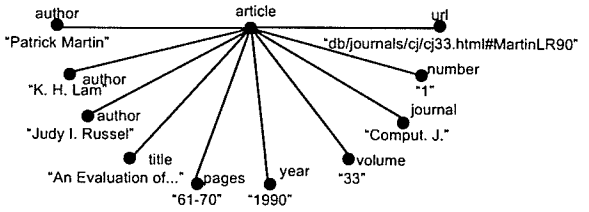


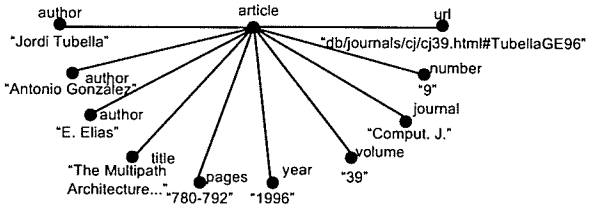
Figure 4.9: Queries of median sizes



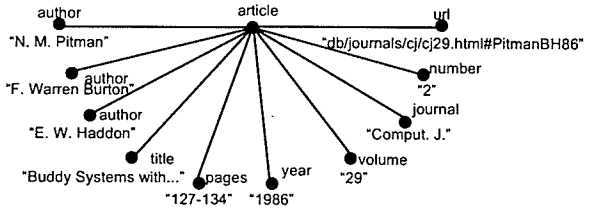
Q11



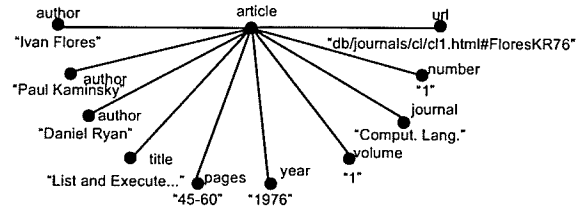
Q12



Q13

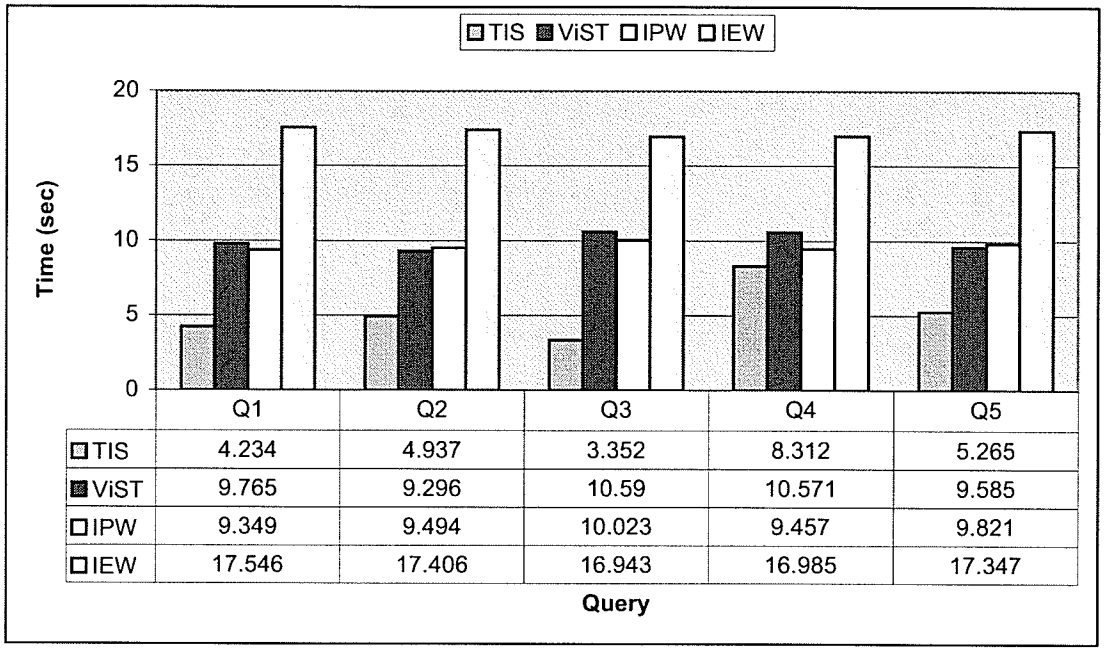


Q14



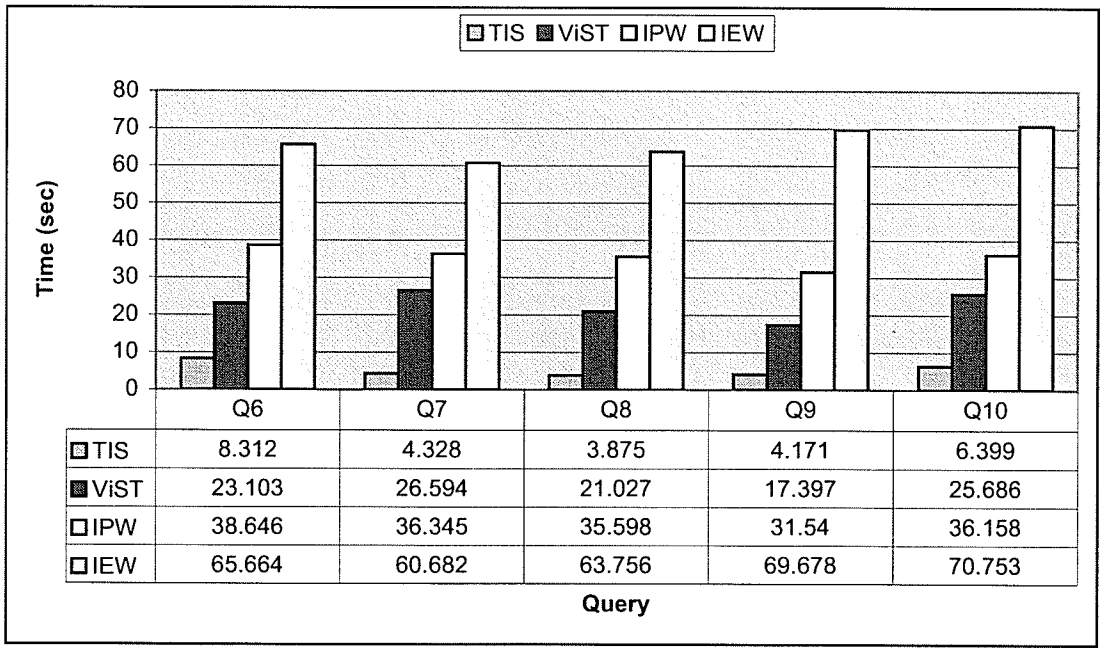
Q15

Figure 4.10: Queries of large sizes



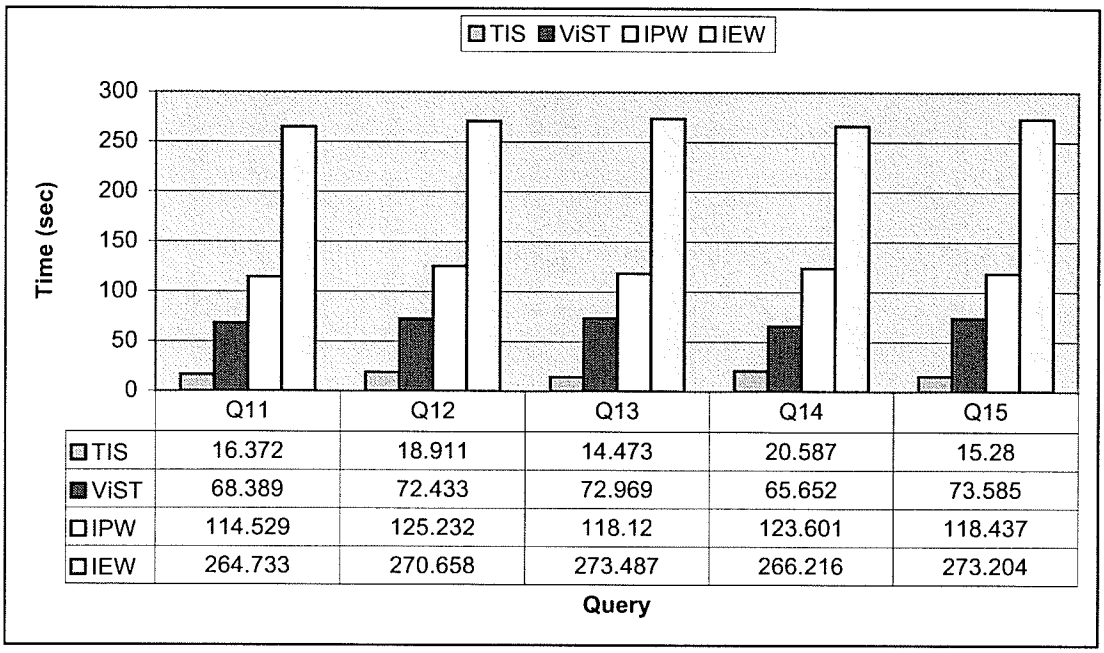
	Q1	Q2	Q3	Q4	Q5
TIS	4.234	4.937	3.352	8.312	5.265
ViST	9.765	9.296	10.59	10.571	9.585
IPW	9.349	9.494	10.023	9.457	9.821
IEW	17.546	17.406	16.943	16.985	17.347

Figure 4.11: Execution time of queries in small sizes



	Q6	Q7	Q8	Q9	Q10
TIS	8.312	4.328	3.875	4.171	6.399
ViST	23.103	26.594	21.027	17.397	25.686
IPW	38.646	36.345	35.598	31.54	36.158
IEW	65.664	60.682	63.756	69.678	70.753

Figure 4.12: Execution time of queries in median sizes



	Q11	Q12	Q13	Q14	Q15
TIS	16.372	18.911	14.473	20.587	15.28
ViST	68.389	72.433	72.969	65.652	73.585
IPW	114.529	125.232	118.12	123.601	118.437
IEW	264.733	270.658	273.487	266.216	273.204

Figure 4.13: Execution time of queries in large sizes

4.7.2 Results

Figure 4.11 to 4.13 show the results of this experiment. From these charts, again, we see that the TIS method beat other methods in this experiment. When queries are small, TIS finished most queries within 6 seconds, whereas IEW took about 17 seconds to finish a query. The performance of IPW and ViST are very close to each other in this category. For median size queries, the average time of evaluating a query by TIS is around 5 seconds, which is almost the same as the average time it used in the small query group. IEW had to spend about 1 minute to finish a query. The performance gap between IPW and ViST widened this time. The average difference between them rose to more than 10 seconds. For large size queries, the average execution time of TIS rose to 17 seconds, but it still performed the best in this group. The average difference between IPW and ViST became 49 seconds. IEW had to spend about 4

minutes more to finish a query.

We found that the performance of TIS appeared to be less stable than the performance of the other methods, especially in the group of small size queries (e.g. Q3, Q4), where its performance variations are more radical than those of large queries (the patterns and sizes of median queries are slightly different). Due to the randomness of signatures, the TIS method may encounter some *false drops* (see Section 1.3.3) during the courses of tree inclusions. If this is the case, the TIS method has to spend extra time to check those false matching documents to identify their matchings, thus, the amount of false drops affects the performance of query evaluation. Q4 seems to be the case. In addition, as we expected, the small queries may be vulnerable to false drops since the one bits in their query signatures are much less than those of large queries.

Overall, TIS did the best in this experiment. The results of TIS suggests that using a signature tree can let the TIS method immediately jump to potential matching documents without running through each document one by one to carry out a tree inclusion check. Since the TIS nearly did not need any time to screen the documents even when the quantity of documents is huge, it definitely outperformed the rest.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis we developed a path-oriented query evaluation method that efficiently retrieves XML documents from a relational database. The motivation for this work was to overcome some limitations of current query evaluation methods, such as the need of expensive database join operations, and the lack of support for optimizations (e.g. signatures). To that end, we combined ideas from the tree inclusion problem and the signature technique in a way that allows us to efficiently evaluate path-oriented queries for searching XML documents from databases.

We presented an overview of the query evaluation problem for XML data, and provided some background information about using both the tree inclusion algorithm and the signature technique to solve this problem. We surveyed literature related to the XML query evaluation and the tree inclusion problem, and we proposed a new top-down tree inclusion algorithm. We discussed the construction of a signature tree to speed up the query evaluation process. We implemented the top-down tree inclusion

algorithm and the signature tree, and we compared the performance of our method to other three methods that are mainly based on index structures. We presented detailed performance results to show that our path-oriented query evaluation method, TIS, outperformed those three methods in most cases.

5.2 Future Work

The following is the potential work for the future development of our query evaluation method.

- **Add attributes**

  In this thesis, we use an XML tree model that only contains two basic components: elements and texts. Sometimes, the attributes in an XML document also have the useful information related to the contents of the document. In the future work, we will use the XML tree model that contains all three basic components to improve the accuracy of query results.

- **Use different RDBMS as repositories**

  Currently, Oracle is the only database system that we used in the experiments. To avoid the situation where experimental results have some potential ties to the Oracle, we will try to use different RDBMS as repositories in our future experiments.

- **Improve signature technique**

  Integrating signature technique with top-down tree inclusion algorithm is just an initiative. As the experiments shows that the signature technique has a

minor issue, i.e. vulnerable to false drops. We will further investigate this issue and make the signature technique perform more reliable.

- **Support XQuery**

  XQuery is a query language that is still under development. In future, it will become a W3C standard of processing many types of XML data sources. We expect many queries will be submitted in XQuery format. The basic building blocks of XQuery are expressions. Therefore, we will need to develop a scheme to handle the mapping between expressions and trees.

Bibliography

[1] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jéôme Siméon. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, October 2004.

[2] Jon Bosak. The plays of Shakespeare in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html, July 1999.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml, October 2000.

[4] Ben Carterette and Fazli Can. Comparing inverted files and signature files for searching a large lexicon. *Inf. Process. Manage.*, 41(3):613–633, 2005.

[5] Weimin Chen. More efficient algorithm for ordered tree inclusion. *J. Algorithms*, 26(2):370–385, 1998.

[6] Yangjun Chen. On the Signature Trees and Balanced Signature Trees. In *Proceedings of International Conference on Data Engineering (ICDE2005)*, pages 742–753. IEEE, April 5 - 8, 2005.

[7] Yangjun Chen and Yong Shi. *Encyclopedia of Database Technologies and Applications*, chapter Signature Files and Signature File Construction, pages 638–645. Idea Group Inc., 2005.

[8] Yangjun Chen and Yong Shi. Tree inclusion algorithm, signatures and evaluation of path-oriented queries. Accepted by 21st Annual ACM Symposium on Applied Computing, Dijon, France, April 23-27, 2006.

[9] James Clark and Steve DeRose. XML path language (XPath) version 1.0. http://www.w3.org/TR/xpath, November 1999.

[10] Uwe Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *SIGIR '86: Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 77–87, New York, NY, USA, 1986. ACM Press.

[11] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A query language for XML. http://www.w3.org/TR/NOTE-xml-ql, August 1998.

[12] C. Faloutsos. *Information Retrieval: Data Structures and Algorithms*, chapter Signature Files, pages 44–65. Prentice Hall, 1992.

[13] Pekka Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases.* PhD thesis, Department of Computer Science, University of Helsinki, November 1992.

[14] Pekka Kilpeläinen and Heikki Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, Reading, MA, 1969.

[16] M. Ley. Computer Science Bibliography. http://www.informatik.uni-trier.de/ ley/db/index.html, 2005.

[17] H. Mannila and K.-J. Räihä. *Information Modelling and Knowledge Bases*, chapter On query languages for the p-string data model, pages 469–482. IOS Press, 1990.

[18] Jonathan Robie, Joe Lapp, and David Schach. XML query language (XQL). http://www.w3.org/TandS/QL/QL98/pp/xql.html, September 1998.

[19] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw Hill, New York, 1983.

[20] T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *ACM SIGIR Workshop On XML and Information Retrieval*, Athens, Greece, July 2000.

[21] Chiyoung Seo, Sangwon Lee, and Hyoung-Joo Kim. An efficient inverted index technique for XML documents using RDBMS. *Elsevier Science B. V.*, 45(1):11–22, January 2003.

[22] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 110–121, New York, NY, USA, 2003. ACM Press.

[23] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 30(2):425–436, 2001.